AD-A041 632 CALIFORNIA UNIV SANTA CRUZ F/G 9/2

BOOTSTRAPPING A SMALL TRANSLATOR WRITING SYSTEM.(U)

MAR 77 M FAY N00014-76-C-0682

UNCLASSIFIED TR-77-3-002 NL

1 OF 1
ADA
041632

END
DATE
FILMED
8-77

BOOTSTRAPPING A SMALL
TRANSLATOR WRITING SYSTEM

by
Michael Fay

Sponsored by: Professor W. M. McKeeman

Technical Report No. 77-3-002

INFORMATION SCIENCES
UNIVERSITY OF CALIFORNIA
SANTA CRUZ, CALIFORNIA 95064

BOOTSTRAPPING A SMALL TRANSLATOR WRITING SYSTEM

by
Michael Fay

Information Sciences
University of California
at
Santa Cruz

March 28, 1977

This research was supported partially by Office of Naval Research Contract No. N00014-76-C-0682

- 1 -

ABSTRACT

A rudimentary translator writing system is developed for easy implementation in about 2 pages of assembly language code. Although the system is based on backtrack parsing and lacks a scanner, it still performs useful translations in a few minutes of CPU time, with storage requirements of about 10K bytes, for a typical translation.

The system is based on an ALGOL-like program by Michels which translates source language strings into target language strings, according to a translation grammar which is specified using prefix Polish operators. Fortunately, the user does not need to specify translation grammars in Polish notation, because Michels gave a metagrammar which translates grammars in BNF-like notation (including the metagrammar itself) into Polish strings.

This report shows how Michels' program can be implemented without the aid of an ALGOL compiler. We present a translation grammar for converting Michels' program (slightly rewritten) into code for a simple, special-purpose interpreter. Once this simple interpreter is implemented, and Michels' program (in interpreter code) and the first input grammar are prepared, a small translator writing system is complete. In this primitive system, a translator "program" consists of the BNF-like description of a translation grammar.

Michels' program was written with the goal of conceptual simplicity. However, in actual performance it was found to be too slow to be practical. Accordingly, we present a new program which is shorter, more efficient, and which requires only a slightly more complex interpreter.

Key words and phrases:

metalanguage, translator, syntax-directed translation, translator writing system, self-describing grammar, interpreter, bootstrapping, backtrack parsing

CR categories: 4.12, 4.13, 4.20

1. Introduction

Programming languages should not interfere with the initial steps in designing programs. Wirth [11] has proposed that programs be written using "stepwise refinement", in which several versions of a program are written in notations convenient to the programmer, but approach some existing target language as the final step. When writing many programs in a single domain (e.g. graphics), the programmer may find a particular intermediate language useful for his needs. If the process of writing translators and interpreters could be simplified, then special-purpose intermediate languages could be implemented, obviating the need for the final refinement steps. Extensions to existing extensible languages could accomplish a similar effect, but would not allow the freedom of language design available to a translator-writer, and would result in a larger, more cumbersome language with unneeded features. Also, extensible languages are generally hard to implement and require large operating systems [1,7,9,10]. In view of the rapid proliferation of smaller computing systems, it would be useful to have simple mechanisms for implementing new special-purpose languages.

In this report, we present a simple implementation of Michels' translator writing system [5], which is based on an earlier paper by Schorre [8]. The system allows simple kinds of translation, directed by a user-produced grammar which is augmented with output symbols. Michels devised metagrammars in notation similar to BNF [6] which define the translation of grammars (including the metagrammars themselves) into strings of prefix Polish operators, suitable for execution on an interpreter. Using ALGOL-like mutually recursive procedures, Michels then presented descriptions of interpreters which would execute a prefix Polish operator string and carry out translations according to the grammar represented by the operator string. The following diagram shows the relationship between a BNF metagrammar, a Polish representation of the same grammar, and an interpreter.

```
------------------      ----------------------      --------------
|       G1       |      | ------------------ |      |  G1-0      |
| A translation  |      | |      G1-0      | |      | An exact   |
| grammar in BNF |      | | A grammar, in  | |      | copy of    |
| for translating|----->| | Polish code,   | |----->| the G1-0   |
| BNF grammars to|      | | for translating| |      | "code"     |
| Polish         |      | | BNF grammars to| |      --------------
------------------      | | Polish         | |
                        | ------------------ |
                        |                    |
                        |       SEEDO        |
                        | Program to scan    |
                        | input and generate |
                        | output according   |
                        | to the operators   |
                        | given in a Polish  |
                        | translation grammar|
                        ----------------------
```

If an arbitrary BNF translation grammar G is used as input, the resulting output will be the Polish form of the grammar (call it G-0). If SEEDO is then executed with G-0 as its program, any input will be recognized and translated according to the rules of G. In this system, G is a simple translator "program". The following diagram illustrates the two-step process involved.

```
 -----------       -------------       -----------
|           |     | ----------- |     |           |
|    G*     |---->||           ||---->|    G-O    |
|   (BNF)   |     ||   G1-O    ||     | (Polish)  |
|           |     || (Polish)  ||     |           |
 -----------      ||           ||      -----------
                  | ----------- |           |
                  |             |           |
                  |    SEEDO    |           |
                  |             |           |
                   -------------            |
                                            |
                         - - - - - - - - - -
                         |
                         v
 -----------       -------------       -----------
|           |     | ----------- |     |           |
|    I*     |     ||           ||     |     O     |
|  Text in  |     ||    G-O    ||     |  Text in  |
| the input |---->||           ||---->|the output |
| language  |     | ----------- |     | language  |
|   of G    |     |             |     |   of G    |
 -----------      |    SEEDO    |      -----------
                  |             |
                   -------------
```

(* indicates typical user-generated input)

This report extends Michels' work by actually implementing the SEEDO interpreter. We do this by translating SEEDO into code for a low-level interpreter. The low-level interpreter is simple enough to be implemented in about 2 pages of PDP-11 assembly language code. Some utility routines and machine-readable texts must also be prepared before the system is complete.

McKeeman [4] calls the system a SEED because even though it is small (and not very powerful and efficient by itself) it can be used as a tool to implement languages for writing scanners, parsers, and other components of more sophisticated translator writing systems. Unfortunately, it turns out that Michels' SEEDO program fails even as a SEED, because it is considerably slower than hand-translation in most cases. Accordingly, we present an implementation of SEEDO only to show the essential ideas needed for a simple translator. We then present a more efficient program, SEED1, consisting of a smaller set of recursive procedures written in a slightly more powerful language. The new language requires new operation codes in the low-level interpreter. However, fewer low-level interpreter

instructions are needed to implement SEED1. In addition, computation time is reduced by a factor of 30 for a typical translation.

2. Formal Definition of Translation

The material in this section, as well as in Section 3, is adapted from Michels [5]

A context-free grammar G is a quadruple (Vt, Vn, S, P) where:

Vt is a finite set of symbols called TERMINALS.
Vn is a finite set of symbols called NON-TERMINALS.
(Vt and Vn are disjoint, and their union is the set V)
S is a distinguished member of Vn called the START or GOAL SYMBOL.
P is a finite set of productions such that each production is a pair (a,b) (alternate notation: a->b). The LEFT PART, a, is a symbol in Vn and the RIGHT PART, b, is a (possibly null) sequence of symbols from V.

> The postfix operator * will denote the set closure or the set of all sequences of symbols in a set. For example, V* represents the set of all strings that can be constructed from the symbols in the alphabet, including the empty string. The operator + denotes the set closure with the exclusion of the empty string.

The set of productions define all possible derivations in T. For all (x,y) in P and u, v in V*, u is derivable from v (written v =>* u) if u can be created by substituting y for any occurrence of x either in v, or in any string derivable from v.

Any string derivable from S is a SENTENTIAL FORM. A sentential form not containing any elements of Vn is a FINAL SENTENTIAL FORM, or SENTENCE.

A TRANSLATION GRAMMAR T is a quintuple (Vi, Vo, Vn, S, P) where Vi and Vo are disjoint sets partitioning Vt. We call Vi the INPUT VOCABULARY and Vo the OUTPUT VOCABULARY. Vn, S, and P are defined as before. T is said to TRANSLATE an element u of Vi* into an element v of Vo* if and only if S =>* z, and deleting the symbols of Vi (respectively, Vo) from z leaves v (respectively, u). Observe that T may translate u into several strings.

Let Pi be P with the symbols from Vo deleted, and Po be P with the symbols from Vi deleted. The INPUT LANGUAGE Li of T is described by the context-free grammar Gi = (Vi, Vn, S, Pi). The OUTPUT LANGUAGE Lo of T is described by the context-free grammar Go = (Vo, Vn, S, Po).

We note in passing that every translation grammar has a "dual" translation grammar in which the roles of Vi and Vo are reversed. Also, we could generalize translation grammars to allow more than two partitions of Vt.

2.1 A Translation Example

We present a grammar that will translate infix expressions to prefix expressions as an example of this class of translations. Consider the translation grammar T = (Vi, Vo, Vn, S, P') where:

Vi = {+,*,a,b}	This is the alphabet of the input language.
Vo = {$\underline{+}$,$\underline{*}$,$\underline{a}$,$\underline{b}$}	This is the object language alphabet; in this case it corresponds one-for-one with Vi. The symbols are underlined to differentiate the two sets.
Vn = {S,T,F,I}	These are the non-terminal symbols.
P' = {S->T, T->$\underline{+}$F+T, T->F, F->$\underline{*}$I*F, F->I, I->$\underline{a}$a, I->$\underline{b}$b}	This is the set of productions defining the translation.

T specifies, among other things, the mapping of 'a+b*a' to '$\underline{+a*ba}$'. The full derivation is as follows:

Sentential Form	Transitional Rule
S	Start symbol
T	S->T
$\underline{+}$F +T	T->$\underline{+}$F+T
$\underline{+}$F +F	T->F
$\underline{+}$F +$\underline{*}$I *F	F->$\underline{*}$I*F
$\underline{+}$I +$\underline{*}$I *I	F->I (used twice)
$\underline{+}$$\underline{a}$a+$\underline{*}$$\underline{b}$b*$\underline{a}$a	I->$\underline{a}$a (twice),I->$\underline{b}$b (final sentential form)
a+ b* a	Input Sentence (symbols from Vo deleted)
$\underline{+a}$ $\underline{*b}$ $\underline{a}$	Output Sentence (symbols from Vi deleted)

3. Translator Implementation

To implement a translator based on a translation grammar, we must create a parser for the input language.

Our assumption here is that the easiest parsing scheme to implement is the top-down backtracking approach. Conceivably, Earley's algorithm [2] or some other method could also be implemented concisely, but this matter is beyond the scope of this paper. The restrictions given in this section apply to our method of backtrack parsing.

To specify the restrictions, we will group together all productions having the same left part. To this end, we use Pi, the set of input productions, to construct the set PLi. Each element of PLi is a list (a,b1,b2,...,bn); <u>a</u> is some left part, and all <u>bi</u> are corresponding right parts. That is, all <u>bi</u> are included in an element of PLi if and only if a->bi is an element of Pi. An input grammar will be represented by a description of PLi, such that the first element is the list in which <u>a</u> is S, the start symbol. A translation grammar is represented by PL, which is PLi with the output symbols restored.

An ordering on the alternative right parts in each element PLi is defined to guarantee that if two right parts can generate input sentences such that one input sentence is a prefix of the other, then the right part generating the longer input sentence is listed first. Formally, for all p in PLi, if b and b' are right parts in p and b precedes b', then for all u, u' elements of Vi+ such that b =>* u and b' =>* u', there exists no u' = ur, where r is in Vi*.

No productions may be empty. That is, for all a->b in Pi, b is in Vi+.

Grammars may not allow left recursion. That is, there may be no v in Vn and u in V* such that v =>* vu .

Deterministic left-to-right parsing is simplified if, whenever a prefix of the remaining input is to be derived from a nonterminal m, the prefix yielding a correct parse is the longest prefix derivable from m. That is, there should not exist any sequence umcv derivable from S, and both x and xc derivable from m, where S is the start symbol, u and v are in (Vi ∪ Vn)*, m is in Vn, and x and c are in Vi+.

The translators to be described in this paper do not detect violations of the restrictions given here; they will merely produce incorrect parses, or fail to terminate at all.

3.1 A Simple Translation Language

A metalanguage can be defined for the syntax and translation of a translation grammar. The notation is similar to BNF [6]. For each non-terminal which is a left

```
R=L "=" A ";" R
 /"END"
 ;
A=[/] C "/" A
 /C
 ;
C=[&] I " " C
 /I
 ;
I="""" (""" """ [#"]/S)
 /"[" ("]]" [>] []]/O)
 /"(" A ")"
 /[:] L
 ;
S=[#] ("]" []]/T) """
 /[&#] ("]" []]/T) S
 ;
O=[>] (""" ["]/T) "]"
 /[&>] (""" ["]/T) O
 ;
L="A" [A]/"B" [B]/"C" [C]/"D" [D]/"E" [E]/"F" [F]/"G" [G]/"H" [H]
 /"I" [I]/"J" [J]/"K" [K]/"L" [L]/"M" [M]/"N" [N]/"O" [O]/"P" [P]
 /"Q" [Q]/"R" [R]/"S" [S]/"T" [T]/"U" [U]/"V" [V]/"W" [W]/"X" [X]
 /"Y" [Y]/"Z" [Z]
 ;
T=L
 /"=" [=]/";" [;]/"(" [(]/")" [)]/"/" [/]/"&" [&]
 /">" [>]/":" [:]/"#" [#]/" " [ ]/"[" [[]/"*" [*]
 /"+" [+]/"-" [-]/"?" [?]/"@" [@]/"," [,]/"$" [$]/"%" [%]
 /"1" [1]/"2" [2]/"3" [3]/"4" [4]/"5" [5]/"6" [6]/"7" [7]
 /"8" [8]/"9" [9]/"0" [0]
 ;
END
```

G1, THE GRAMMAR GRAMMAR

FIGURE 1

part of a production, all right parts for that non-terminal are listed, separated by a slash (/). The left part will be separated from the alternative right parts by an equal sign (=). Juxtaposition will denote the concatenation of elements of a right part. Double quotes (") will delimit elements in Vi+. Brackets ([,]) will be used to delimit elements of Vo+. Single letters denote elements of Vn. Normal parentheses can be used to alter the implied operator precedence and to reduce the number of productions required by allowing the factoring of rules.

Fig. 1 shows G1, the translation grammar describing the "grammar language", expressed in its own language. It is a modification of the grammar given in Section 4.1 of Michels [5]. (Productions I, S, and O were modified so as to minimize backtracking in most cases, and L was split into L and T, with many new symbols added). Multiple blanks and ends of lines have no meaning in the language. They have been used here to improve readability and should be ignored.

Literal strings may be of arbitrary length. This creates a problem if a string must contain a double quote ("), which is the literal delimiter. To solve this the production for "I" (see Fig. 1) is ordered to test for a double quote as the first character of a literal string; if one is found it is assumed to be the entire string and must be followed by the terminating double quote. A double quote in any other position of a literal string is assumed to be the terminating delimiter of that string. A similar convention is used to denote right bracket (]) as an output symbol.

The self-translation of G1, giving the "object grammar" G1-O, is shown in Fig. 2. Paragraphing has been added to improve readability. The actual machine language, as defined by the translator and accepted by the machine, would be a continuous string of characters. The only significant blank is one which follows an odd number of sharps (#). The next section explains the meaning of #.

3.2 The Object Language

The output of the metatranslator described in Section 3.1 is a description of a translation grammar. This output can be interpreted by the program to be described in the next section, and can be thought of as an "object language for grammars". This language contains five prefix operators. The '&' is a binary concatenation operator; it has the value TRUE if and only if both of the operands which follow it are TRUE. The '/' is the binary alternation operator, which has the value TRUE if either of the operands following it are TRUE. If the first is TRUE the second is

```
R/&:L&#=&:A&#;:R
  &#E&#N#D
A/&>/&:C&#/:A
  :C
C/&>&&:I&# :C
  :I
I/&#"
    /&#"
      &#"&>#>"
     :S
 /&#[
    /&&#]#]
      &>>>]
     :O
 /&#(&:A#)
   &>::L
S/&>#
    &/&#]>]
      :T
     #"
   &&>&>#
    &/&#]>]
      :T
     :S
O/&>>
    &/&#">"
      :T
     #]
   &&>&>>
    &/&#">"
      :T
     :O
L/&#A>A/&#B>B/&#C>C/&#D>D/&#E>E/&#F>F/&#G>G/&#H>H/&#I>I/&#J>J
 /&#K>K/&#L>L/&#M>M/&#N>N/&#O>O/&#P>P/&#Q>Q/&#R>R/&#S>S/&#T>T
 /&#U>U/&#V>V/&#W>W/&#X>X/&#Y>Y&#Z>Z
T/:L/&#=>=/&#;>;/&#(>(/&#)>)/&#/>//&#&>&/&#>>>/&#:>:
 /&##>#/&# > /&#[>[/&#*>*
 /&#+>+/&#->-/&#?>?/&#@>@/&#,>,/&#$>$/&#%>%
 /&#1>1/&#2>2/&#3>3/&#4>4/&#5>5/&#6>6/&#7>7
 /&#8>8/&#9>9&#0>0
```

G1-O, THE OBJECT VERSION OF G1

FIGURE 2

not tested. If the first is FALSE, both input and output strings are restored to their pre-test value before testing the second. The ':' is a unary non-terminal operator; it has the value TRUE if the rule labeled by its operand is TRUE. The '#' is a unary terminal operator; it has the value TRUE if the current character of input is the same as the operand, in which case the input is advanced one character. The '>' is a unary operator and always has the value TRUE. The character following it is appended to the end of the current output string.

3.3 The Translator Interpreter

We now define an interpreter to execute the object code emitted by the translator of Section 3.1. The interpreter is presented as a set of mutually recursive functions in the ALGOL-like language SEEDGOL-0, to be described here and in Section 4.

SEEDGOL-0 has three types of data: strings, single characters, and boolean values. All strings are substrings (in fact, "tails") of either the object grammar G, or the source input I, both of which are inputs to any SEEDGOL-0 program. We refer to the sets of tails of G and I as STRINGS(G) and STRINGS(I), respectively.

The built-in functions of SEEDGOL-0 are as follows:

First: STRINGS -> CHARACTERS
First(S) is the first character of the string S.

Rest: STRINGS -> STRINGS
Rest(S) is all but the first character of S.

Output: CHARACTERS ->
Output(C) has no value, but causes the side effect of sending C to an output device or buffer.

Equal: CHARACTERS x CHARACTERS -> BOOLEANS
Equal(C1,C2) is TRUE if and only if C1 and C2 are identical characters.

Isnulli: STRINGS(I) -> BOOLEANS
Isnulli(S) is TRUE if and only if S is null.

The built-in operation of pairing any two expressions is also allowed, by using parentheses. The constants TRUE and FALSE are included in the language.

We now give the functionality and meaning of the interpreter definition functions, implemented in the SEEDGOL-0 program SEED0 (shown in Fig. 3).

```
IF TEST(REST(GP),IP) THEN
   IF ISNULLI(REMAINING(REST(GP),IP)) THEN
      (TRUE, EMIT(REST(GP), IP))
   ELSE (FALSE, NULL)
ELSE (FALSE, NULL)
;
DEFINE TEST(RP, IP) =
   IF EQUAL(FIRST(RP), ":") THEN TEST(FIND(GP, REST(RP)), IP)
   ELSE IF EQUAL(FIRST(RP), "&") THEN
      IF TEST(REST(RP), IP) THEN
         TEST(SKIP(REST(RP), IP), REMAINING(REST(RP), IP))
         ELSE FALSE
   ELSE IF EQUAL(FIRST(RP), "/") THEN
      IF TEST(REST(RP), IP) THEN TRUE
      ELSE TEST(SKIP(REST(RP), IP), IP)
   ELSE IF EQUAL(FIRST(RP), ">") THEN TRUE
   ELSE EQUAL(FIRST(REST(RP)), FIRST(IP));

DEFINE REMAINING(RP, IP) =
   IF EQUAL(FIRST(RP), ":") THEN REMAINING(FIND(GP, REST(RP)), IP)
   ELSE IF EQUAL(FIRST(RP), "&") THEN
      REMAINING(SKIP(REST(RP), IP), REMAINING(REST(RP), IP))
   ELSE IF EQUAL(FIRST(RP), "/") THEN
      IF TEST(REST(RP), IP) THEN REMAINING(REST(RP), IP)
      ELSE REMAINING(SKIP(REST(RP), IP), IP)
   ELSE IF EQUAL(FIRST(RP), ">") THEN IP
   ELSE REST(IP);

DEFINE EMIT(RP, IP) =
   IF EQUAL(FIRST(RP), ":") THEN EMIT(FIND(GP, REST(RP)), IP)
   ELSE IF EQUAL(FIRST(RP), "&") THEN
      (EMIT(REST(RP), IP),
            EMIT(SKIP(REST(RP), IP), REMAINING(REST(RP), IP)))
   ELSE IF EQUAL(FIRST(RP), "/") THEN
      IF TEST(REST(RP), IP) THEN EMIT(REST(RP), IP)
      ELSE EMIT(SKIP(REST(RP), IP), IP)
   ELSE IF EQUAL(FIRST(RP), ">") THEN OUTPUT(FIRST(REST(RP)))
   ELSE NULL;

DEFINE SKIP(RP, IP) =
   IF EQUAL(FIRST(RP), "&") THEN SKIP(SKIP(REST(RP), IP), IP)
   ELSE IF EQUAL(FIRST(RP), "/") THEN SKIP(SKIP(REST(RP), IP), IP)
   ELSE REST(REST(RP));

DEFINE FIND(RP, IP) =
   IF EQUAL(FIRST(RP), FIRST(IP)) THEN REST(RP)
   ELSE FIND(SKIP(REST(RP), IP), IP);
END
```

SEEDO

FIGURE 3

SEEDO: STRINGS(G) x STRINGS(I) -> BOOLEANS
SEEDO(G,I) is TRUE if and only if the input I is recognized by the string of prefix operators G. As a side effect, SEEDO causes the proper characters to be output.

Test: STRINGS(G) x STRINGS(I) -> BOOLEANS
Test(RULE,INPUT) is TRUE if any left-most substring of INPUT matched by RULE.

Remaining: STRINGS(G) x STRINGS(I) -> STRINGS(I)
Remaining(RULE,INPUT) is the substring of INPUT remaining after the substring recognized by RULE is removed.

Emit: STRINGS(G) x STRINGS(I) ->
Emit(RULE,INPUT) has no value, but causes output characters to be sent to an output device or buffer while INPUT is recognized by RULE.

Skip: STRINGS(G) x STRINGS(I) -> STRINGS(G)
Skip(RULE,INPUT) is the substring of RULE remaining after the leftmost operator and its operands have been removed. INPUT has no bearing on the computation, but is required by syntax.

Find: STRINGS(G) x STRINGS(G) -> STRINGS(G)
Find(GRAMMAR,STRING) is the substring of GRAMMAR labeled by the first character of STRING.

4. Implementing SEEDGOL-0

To implement the SEEDGOL-0 language, we will produce a translation grammar for converting SEEDGOL-0 programs into intermediate code, and we will produce an interpreter for executing the code. We will refer to this low-level interpreter, M0, as a "machine", to avoid confusing it with the SEEDO program, an interpreter for object grammars.

The SEEDGOL-0 language will be very specialized, containing merely the constructs needed to implement the SEEDO interpreter in Fig. 3. The following restrictions apply to the SEEDGOL-0 language:

1. All blanks are ignored. In general, a reserved word should not be a prefix of an identifier or another reserved word.

2. There are no declarations other than procedure declarations. The only variables are the parameters RP and IP, which are strings. At the outermost level, IP is predefined to be all of the input string, and RP is predefined to be same as GP. GP is a constant: the object grammar string, a sequence of prefix operators.

3. All procedures have exactly 2 parameters. Pairs and procedure arguments are evaluated left-to-right.

4. Procedures and the mainline each consist of one expression.

5. The language has no I/O facility other than the built-in "Output" function, which is implementation dependent. Initialization of the input and grammar string storage areas is also implementation dependent, and is assumed to be completed before the SEEDGOL-0 machine begins execution.

4.1 Translating SEEDGOL-0 into Object Code

Fig. 4-A shows SGLOG (<u>S</u>EED<u>G</u>O<u>L</u>-<u>0</u> <u>G</u>rammar), a grammar for describing SEEDGOL-0 and for translating it into a mnemonic form of machine language of M0, the SEEDGOL-0 machine (to be described in the next section). The output strings of SGLOG are either instructions, characters, or numeric constants. Fig. 4-B shows the same SGLOG grammar, with outputs represented in a form which can be more easily interpreted by a computer program. The occurrence of "I" means that the following base-10 number will represent an instruction, while a double quote (") means that the next character is a character constant. A "D" precedes a base-10 numerical constant. Fig. 5-B shows SEED0-0, which is the result of translating SEED0 (Fig. 3) via the translation grammar SGLOG (Fig. 4-B). SEED0-0 is a form of the "object code" for SEED0 on M0. A mnemonic version of SEED0-0 is given in Fig. 5-A.

```
P=E ";" [RETURN] D;
E="IF" E [IF] "THEN" E [THEN] "ELSE" E [ELSE]
 /"EQUAL(" E "," E ")" [EQUAL]
 /"ISNULLI(" E ")" [ISNULLI]
 /"REST(" E ")" [REST]
 /"OUTPUT(" E ")" [OUTPUT]
 /"FIRST(" E ")" [FIRST]
 /"TRUE" [TRUE]/"FALSE" [FALSE]/"NULL"
 /"GP" [GP]/"RP" [RP]/"IP" [IP]
 /[PROCN] I "(" E "," E ")" [CALL]
 /"(" E "," E ")"
 /""" S """
 ;
D="DEFINE" I "(RP,IP)=" E ";" [RETURN] D
 /"END"
 ;
I=L I
 /L [ ]
 ;
S="!" [!]/"&" [&]/"/" [/]/">" [>]/"" [#];
L="A" [A]/"B" [B]/"C" [C]/"D" [D]/"E" [E]/"F" [F]/"G" [G]
 /"H" [H]
 /"I" [I]/"J" [J]/"K" [K]/"L" [L]/"M" [M]/"N" [N]/"O" [O]
 /"P" [P]
 /"Q" [Q]/"R" [R]/"S" [S]/"T" [T]/"U" [U]/"V" [V]/"W" [W]
 /"X" [X]
 /"Y" [Y]/"Z" [Z]
 ;
 END
```

SGLOG, THE SEEDGOL-0 TRANSLATION GRAMMAR
(MNEMONIC VERSION)

FIGURE 4-A

```
P=E ";" [I13] D;
E="IF" E [I8] "THEN" E [I9] "ELSE" E [I10]
 /"EQUAL(" E "," E ")" [I7]
 /"ISNULLI(" E ")" [I6]
 /"REST(" E ")" [I4]
 /"OUTPUT(" E ")" [I5]
 /"FIRST(" E ")" [I3]
 /"TRUE" [D1]/"FALSE" [D0]/"NULL"
 /"QP" [I0]/"RP" [I1]/"IP" [I2]
 /[I11] I "(" E "," E ")" [I12]
 /"(" E "," E ")"
 /""" S """
 ;
D="DEFINE" I "(RP,IP)=" E ";" [I13] D
 /"END"
 ;
I=L I
 /L [" ]
 ;
S="!" ["!]/"&" ["&]/"/" ["/]/">" [">]/"#" ["#];
L="A" ["A]/"B" ["B]/"C" ["C]/"D" ["D]/"E" ["E]/"F" ["F]/"G" ["G]
 /"H" ["H]/"I" ["I]/"J" ["J]/"K" ["K]/"L" ["L]/"M" ["M]/"N" ["N]
 /"O" ["O]/"P" ["P]/"Q" ["Q]/"R" ["R]/"S" ["S]/"T" ["T]/"U" ["U]
 /"V" ["V]/"W" ["W]/"X" ["X]/"Y" ["Y]/"Z" ["Z]
 ;
END
```

SGLOG, THE SEEDGOL-0 TRANSLATION GRAMMAR
(MACHINE VERSION)

FIGURE 4-B

001 TO 010:	PROCN	T	E	S	T		GP	REST	IP	CALL
011 TO 020:	IF	PROCN	R	E	M	A	I	N	I	N
021 TO 030:	G		GP	REST	IP	CALL	ISNULL	IF	1	PROCN
031 TO 040:	E	M	I	T		GP	REST	IP	CALL	THEN
041 TO 050:	0	ELSE	THEN	0	ELSE	RETURN	T	E	S	T
051 TO 060:		RP	FIRST	:	EQUAL	IF	PROCN	T	E	S
061 TO 070:	T		PROCN	F	I	N	D		GP	RP
071 TO 080:	REST	CALL	IP	CALL	THEN	RP	FIRST	&	EQUAL	IF
081 TO 090:	PROCN	T	E	S	T		RP	REST	IP	CALL
091 TO 100:	IF	PROCN	T	E	S	T		PROCN	S	K
101 TO 110:	I	P		RP	REST	IP	CALL	PROCN	R	E
111 TO 120:	M	A	I	N	I	N	G		RP	REST
121 TO 130:	IP	CALL	CALL	THEN	0	ELSE	THEN	RP	FIRST	/
131 TO 140:	EQUAL	IF	PROCN	T	E	S	T		RP	REST
141 TO 150:	IP	CALL	IF	1	THEN	PROCN	T	E	S	T
151 TO 160:		PROCN	S	K	I	P		RP	REST	IP
161 TO 170:	CALL	IP	CALL	ELSE	THEN	RP	FIRST	>	EQUAL	IF
171 TO 180:	1	THEN	RP	REST	FIRST	IP	FIRST	EQUAL	ELSE	ELSE
181 TO 190:	ELSE	ELSE	RETURN	R	E	M	A	I	N	I
191 TO 200:	N	G		RP	FIRST	:	EQUAL	IF	PROCN	R
201 TO 210:	E	M	A	I	N	I	N	G		PROCN
211 TO 220:	F	I	N	D		GP	RP	REST	CALL	IP
221 TO 230:	CALL	THEN	RP	FIRST	&	EQUAL	IF	PROCN	R	E
231 TO 240:	M	A	I	N	I	N	G		PROCN	S
241 TO 250:	K	I	P		RP	REST	IP	CALL	PROCN	R
251 TO 260:	E	M	A	I	N	I	N	G		RP
261 TO 270:	REST	IP	CALL	CALL	THEN	RP	FIRST	/	EQUAL	IF
271 TO 280:	PROCN	T	E	S	T		RP	REST	IP	CALL
281 TO 290:	IF	PROCN	R	E	M	A	I	N	I	N
291 TO 300:	G		RP	REST	IP	CALL	THEN	PROCN	R	E
301 TO 310:	M	A	I	N	I	N	G		PROCN	S
311 TO 320:	K	I	P		RP	REST	IP	CALL	IP	CALL
321 TO 330:	ELSE	THEN	RP	FIRST	>	EQUAL	IF	IP	THEN	IP
331 TO 340:	REST	ELSE	ELSE	ELSE	ELSE	RETURN	E	M	I	T
341 TO 350:		RP	FIRST	:	EQUAL	IF	PROCN	E	M	I
351 TO 360:	T		PROCN	F	I	N	D		GP	RP
361 TO 370:	REST	CALL	IP	CALL	THEN	RP	FIRST	&	EQUAL	IF
371 TO 380:	PROCN	E	M	I	T		RP	REST	IP	CALL
381 TO 390:	PROCN	E	M	I	T		PROCN	S	K	I
391 TO 400:	P		RP	REST	IP	CALL	PROCN	R	E	M
401 TO 410:	A	I	N	I	N	G		RP	REST	IP
411 TO 420:	CALL	CALL	THEN	RP	FIRST	/	EQUAL	IF	PROCN	T
421 TO 430:	E	S	T		RP	REST	IP	CALL	IF	PROCN
431 TO 440:	E	M	I	T		RP	REST	IP	CALL	THEN
441 TO 450:	PROCN	E	M	I	T		PROCN	S	K	I
451 TO 460:	P		RP	REST	IP	CALL	IP	CALL	ELSE	THEN
461 TO 470:	RP	FIRST	>	EQUAL	IF	RP	REST	FIRST	OUTPUT	THEN
471 TO 480:	ELSE	ELSE	ELSE	ELSE	RETURN	S	K	I	P	
481 TO 490:	RP	FIRST	&	EQUAL	IF	PROCN	S	K	I	P
491 TO 500:		PROCN	S	K	I	P		RP	REST	IP
501 TO 510:	CALL	IP	CALL	THEN	RP	FIRST	/	EQUAL	IF	PROCN
511 TO 520:	S	K	I	P		PROCN	S	K	I	P
521 TO 530:		RP	REST	IP	CALL	IP	CALL	THEN	RP	REST
531 TO 540:	REST	ELSE	ELSE	RETURN	F	I	N	D		RP
541 TO 550:	FIRST	IP	FIRST	EQUAL	IF	RP	REST	THEN	PROCN	F
551 TO 560:	I	N	D		PROCN	S	K	I	P	
561 TO 570:	RP	REST	IP	CALL	IP	CALL	ELSE	RETURN		

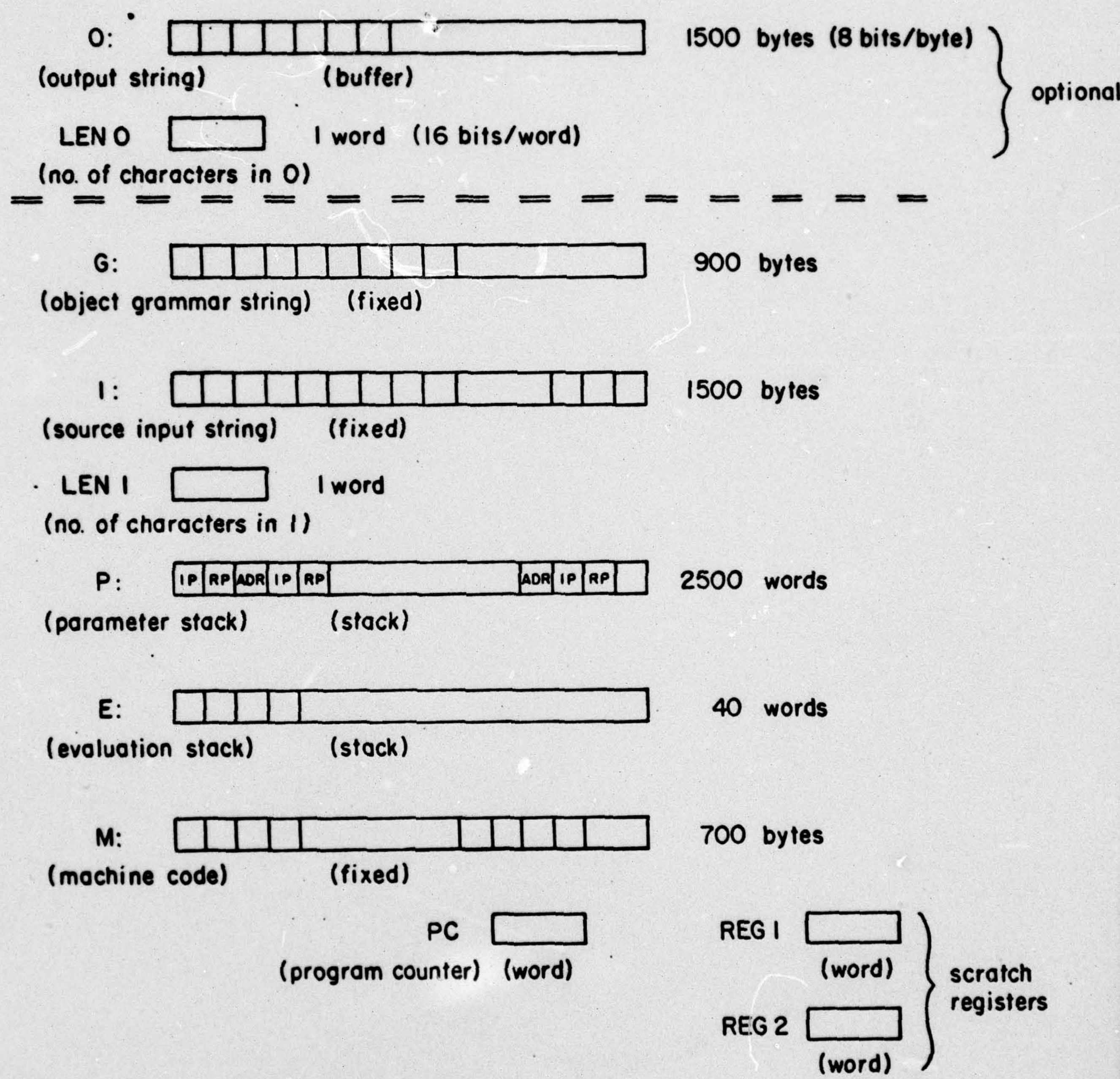
SEEDO-O, THE MO OBJECT CODE VERSION OF SEEDO
(MNEMONIC VERSION)

FIGURE 5-A

```
I11"T"E"S"T" IOI4I2I1I2I8I11"R"E"M"A"I"N"I"N"G" IOI4I2I1I2I6I8D1I11"E"M"I"
T" IOI4I2I1I2I9DOI1OI9DOI1OI13"T"E"S"T" I1I3"!I7I8I11"T"E"S"T" I11"F"I"N"
D" IOI1I4I1I2I2I1I2I9I1I3"&I7I8I11"T"E"S"T" I1I4I2I1I2I8I11"T"E"S"T" I11"S"
K"I"P" I1I4I2I1I2I11"R"E"M"A"I"N"I"N"G" I1I4I2I1I2I1I2I9DOI1OI9I1I3"/I7I8I1
1"T"E"S"T" I1I4I2I1I2I8D1I9I11"T"E"S"T" I11"S"K"I"P" I1I4I2I1I2I2I1I2I1OI9I
1I3">I7I8D1I9I1I4I3I2I3I7I1OI1OI1OI1OI13"R"E"M"A"I"N"I"N"G" I1I3"!I7I8I1
1"R"E"M"A"I"N"I"N"G" I11"F"I"N"D" IOI1I4I1I2I2I1I2I9I1I3"&I7I8I11"R"E"M"A"
I"N"I"N"G" I11"S"K"I"P" I1I4I2I1I2I11"R"E"M"A"I"N"I"N"G" I1I4I2I1I2I1I2I9I1
I3"/I7I8I11"T"E"S"T" I1I4I2I1I2I8I11"R"E"M"A"I"N"I"N"G" I1I4I2I1I2I9I11"R"
E"M"A"I"N"I"N"G" I11"S"K"I"P" I1I4I2I1I2I2I1I2I1OI9I1I3">I7I8I2I9I2I4I1OI1
OI1OI1OI13"E"M"I"T" I1I3"!I7I8I11"E"M"I"T" I11"F"I"N"D" IOI1I4I1I2I2I1I2I9
I1I3"&I7I8I11"E"M"I"T" I1I4I2I1I2I11"E"M"I"T" I11"S"K"I"P" I1I4I2I1I2I11"R
"E"M"A"I"N"I"N"G" I1I4I2I1I2I1I2I9I1I3"/I7I8I11"T"E"S"T" I1I4I2I1I2I8I11"E"
M"I"T" I1I4I2I1I2I9I11"E"M"I"T" I11"S"K"I"P" I1I4I2I1I2I2I1I2I1OI9I1I3">I7I
8I1I4I3I5I9I1OI1OI1OI1OI13"S"K"I"P" I1I3"&I7I8I11"S"K"I"P" I11"S"K"I"P"
I1I4I2I1I2I2I1I2I9I1I3"/I7I8I11"S"K"I"P" I11"S"K"I"P" I1I4I2I1I2I2I1I2I9I1I4
I4I1OI1OI13"F"I"N"D" I1I3I2I3I7I8I1I4I9I11"F"I"N"D" I11"S"K"I"P" I1I4I2I
1I2I2I1I2I1OI13
```

SEEDO-O, THE MO OBJECT CODE VERSION OF SEEDO
(MACHINE VERSION)

FIGURE 5-B



MØ, The SEEDGOL-Ø Machine
(with typical buffer sizes shown)

Fig. 6

```
OPCODE   MNEMONIC    DESCRIPTION
------   --------    ------------------------------------------
 (ANY DATA ITEM)     (ITEM)  PUSH THE ITEM.
  0      GP          (STRING) PUSH POINTER TO FIRST CHARACTER IN G.

  1      RP          (STRING) PUSH RP PARAMETER (STORED AT THE
                       TOP OF THE P STACK).

  2      IP          (STRING) PUSH IP PARAMETER (STORED NEXT TO
                       THE TOP OF THE P STACK).

  3      FIRST       (STRING => CHAR)  REPLACE TOP OF E WITH THE
                       CHARACTER TO WHICH IT POINTS.

  4      REST        (STRING => STRING) INCREMENT PTR AT TOP OF E.

  5      OUTPUT      (CHAR =>)  POP TOP OF E TO OUTPUT.

  6      ISNULLI     (STRING(I) => BOOLEAN) REPLACE TOP OF E WITH
                        TRUE IF IT POINTS TO END OF I, ELSE FALSE.

  7      EQUAL       (ITEM x ITEM => BOOLEAN)  REPLACE THE TOP 2
                        ITEMS OF E WITH TRUE IF EQUAL, ELSE FALSE.

  8      IF          (ITEM => )  POP E.  IF FALSE, SET PC TO LOC OF
                        NEXT MATCHING "THEN" + 1.

  9      THEN        ( )  SKIP TO NEXT MATCHING "ELSE" + 1.

 10      ELSE        ( )  NO OPERATION; SERVES AS MARKER ONLY.

 11      PROCN       (ADDR)  COMPARE THE CHARACTERS IN M AT PC
                       WITH ALL STRINGS FOLLOWING RETURN INSTR"S
                       ANYWHERE IN M.  (BLANK IS ALWAYS THE FINAL
                       CHARACTER.)  PUSH THE LOCATION IMMEDIATELY
                       AFTER THE MATCHING STRING (THIS
                       IS THE CALLED ADDRESS).

 12      CALL        (ADDR x STRING x STRING => )  PUSH PC (RETURN
                       ADDRESS) ONTO P, THEN PUSH POP(E) ONTO P
                       TWICE, THEREBY PASSING THE IP AND RP
                       PARAMETERS.  THEN SET PC = POP(E), THE CALLED
                       ADDRESS (COMPUTED AND STACKED BY AN EARLIER
                       PROCN INSTRUCTION).

 13      RETURN      ( )  POP P TWICE.  IF EMPTY, SET HALT
                        CONDITION, ELSE SET PC = POP(P), RETURNING.
```

ALL PUSHES AND POPS REFER TO THE E STACK, EXCEPT AS NOTED.
CONSTANT DATA ITEMS ARE DISTINGUISHABLE FROM INSTRUCTIONS.
THE SET "ITEM" IS THE UNION OF CHAR, STRING, AND BOOLEAN.

MO MACHINE INSTRUCTIONS

FIGURE 7

4.2 The SEEDGOL-0 Machine

A diagram of the SEEDGOL-0 machine, M0, is given in Fig. 6. M0 contains two string storage areas--one for the object grammar input (G), and one for the input string to be recognized and translated (I). Output is presented serially to a device or to a storage buffer. There is an evaluation stack, E, which holds all operands. The machine code contains operators which affect the contents of E. Since SEEDGOL-0 allows recursion, there is a parameter stack, P, which contains, for each call of a procedure, two parameters (called RP and IP), and a return address. The lowest level of P corresponds to the mainline, for which there is no return address. The machine code is stored in an area called M, with PC pointing to the next instruction to be executed. M contains both instructions and constant data. The contents of the areas G, I, and M do not change during execution; the contents of the stacks E, P, and O, as well as the temporary registers REG1 and REG2 and the program counter PC, do ordinarily change during execution.

The only strings which can be referred to by variables in a SEEDGOL-0 program are tails of two fixed strings: the grammar (G) and the input (I). As a result, string variables can be represented by pointers to locations in G or I. The string represented by a pointer consists of all characters in G or I beginning with the character pointed to.

In the machine code storage area M, constants may have any representation which is easily distinguished from the instructions. As a matter of convenience, the implementation in this report uses 8-bit representations of instructions and constants, with the high-order bit set to 0 for constants and 1 for instructions. The seven remaining bits are sufficient to represent the 14 instruction codes and 34 constants which can be produced by SGLOG. Included among the constants are the booleans FALSE (represented here by 0), TRUE (1), and alphabetic and other characters (represented by ASCII or some other 7-bit code). The representation of constants is specified by the grammar SGLOG (Fig. 4-B). A description of the machine code operators is given in Fig. 7. All unary and binary operators expect their operands on E. Most instructions are quite simple. However, IF, THEN, ELSE, PROCN, CALL, and RETURN are interesting enough to warrant examples.

We first describe an example of an IF-THEN-ELSE expression, Referring to the grammar SGLOG in Fig. 4-B, one can see that the IF expression

IF (exp1) THEN (exp2) ELSE (exp3)

will be translated into

(exp1*) IF (exp2*) THEN (exp3*) ELSE,

where the starred expressions are the machine code versions of the source expressions, and capitalized words in the second line denote machine instructions. Normally, (exp1*) causes a single boolean value to be pushed onto the evaluation stack, E. (A pathological SEEDGOL-0 program could have NULL or an ordered pair as (exp1), thereby causing zero or two parameters to be pushed on E. This would cause the machine to fail.) The IF operator pops this value off E, and transfers program control to the next matching THEN + 1 if the value is FALSE. Otherwise, execution continues at the next instruction (i.e., exp2* is executed). The "then" part of the if expression is exp2*, delimited by the IF and THEN instructions. The "else" part of the if expression is exp3*, delimited by THEN and ELSE. When a THEN is executed, the program counter is unconditionally advanced to the next matching ELSE + 1, and an ELSE, when executed, is a no-op. ELSE is merely a marker used by the THEN instruction.

We now consider the mechanics of procedure calls in M0. Suppose that the SEEDGOL-0 source specifies the invocation of the procedure JUNK, as follows:

JUNK(exp1, exp2) .

This will be translated into

PROCN J U N K (exp1*) (exp2*) CALL .

When the PROCN instruction is executed, the entire machine code program is searched for an occurrence of a RETURN instruction, followed by the characters J U N K . (Since all procedures must follow other procedures or the mainline, and all procedures have the characters of their name first, we see that the only place to look for procedure names as destinations is immediately following any RETURN instruction.) PROCN thus determines the called address, and pushes it on E. Next, two expressions, the actual parameters, cause two values to be pushed on E. CALL then pushes three items onto P: the contents of PC (i.e. the return address), and the actual values of the two parameters (these values are popped from E; their order on P is reversed). Finally, CALL transfers control to the called procedure by popping E once more and storing this value (the called address) in PC. At the end of a procedure, a RETURN instruction is executed. It pops the top two parameters and

the return address from P, transferring control to the return address.

B5700 Extended ALGOL and PDP-11 assembly language descriptions of M0 are given in Appendices A and B.

4.3 Initial Bootstrapping

To create a simple translator-writing system, the user must:

1. Transcribe G1-0 (Fig. 2), the "object" grammar grammar into machine readable form.

2. Transcribe SEEDO-0 (Fig. 5-B), the object code for SEEDO on M0, into machine readable form.

3. Implement M0 itself (Appendices A and B are examples).

4. Implement a means for getting prefix Polish "object" grammars, M0 instructions, and input strings into the storage areas for M0, and a means to retrieve the output string after translation.

Once these four steps are complete, a grammar submitted as input will be translated into its object form, and this object grammar can replace G1-0 to give a new translator. This process is represented by the second diagram in Section 1. (One can also describe such a series of translations in a functional notation similar to GENESIS [3], a language used to describe sequences of program runs.)

5. An Improved Interpreter

It was originally hoped that the SEEDO program, although clearly inefficient, would still be able to translate grammars in a reasonable amount of time. It was to be used only for a few simple translations before being replaced by a more efficient translator. However, we estimate that SEEDO running on the PDP-11 assembly language version of M0 will take 6 1/2 hours to translate G1 to G1-0. Hand translation would be faster. The reason that SEEDO and M0 have been presented at all is that they provide a conceptually simple description of a basic translator writing system. We shall see that a practical TWS can be obtained by extending this simple one.

5.1 An Analysis of SEEDO Execution Time

A casual study of the SEEDO program (Fig. 3) shows great inefficiencies. The primary observation is that once Test(RP,IP) is computed, not only is a prefix of IP accepted or rejected, but the length of the prefix is known, and the output symbols encountered in RP during the recognition of IP are also known. If we could make this information available to the mainline program on the outermost call of TEST, we could reduce computation time by a factor of 3, since Remaining and Emit, both às time-consuming as Test, would not need to be invoked. (Note that Emit would then become totally unnecessary.)

More significantly, we could eliminate the call of Remaining from within Test. This occurs in the following context within Test:

```
Test(RP,IP) = ...
... IF Equal(First(RP),"&") THEN
        Test(Skip(Rest(RP),IP), Remaining(Rest(RP), IP))
    ELSE FALSE
...
```

Suppose the G and I strings are as follows:

... & (A) (B) ...	...aaaaaabbbbbbbb...
G	I

(A) and (B) are strings which comprise the two operands of &. Suppose RP begins with the &. Then Rest(RP) begins at the head of the substring (A), while Skip(Rest(RP), IP) begins at the head of the substring (B). We first use Rest(RP) to recognize a prefix of IP (the a's above, say), and if successful, we use Skip(Rest(RP), IP) to recognize part of the remainder of IP (represented by the b's above). Remaining(Rest(RP),IP) calculates this remainder; we do not retain the information about the length of the prefix of IP (the a's above) recognized during the call of Test(Rest(RP),IP).

Let n be the level of recursive calls of Test with First(RP) = "&" (i.e. n equals the depth of recursion in Test, minus those calls which do not have First(RP) = "&"). Since Remaining can call Test (if First(RP) = "/") with about the same likelihood that Test can call Remaining, let us assume that all Remaining and Test calls made at the same n-level require the same amount of time, T(n-1). Then we estimate that:

T(n) = 3T(n-1), i.e. T(n) = 3**n, if T(0) = 1.

But if Test's call to Remaining were removed, then:

T(n) = 2T(n-1), i.e. T(n) = 2**n, if T(0) = 1.

Even though the majority of calls of Test and Remaining would not take advantage of this savings, it is still significant, because it grows exponentially with the depth of recursion, which can get quite large (e.g. about 200 for G1, with n approaching 30 or so), suggesting that a substantial speedup is possible. A new program could carry out a large part of the computation done by SEED0 in roughly (2/3)**n the time.

5.2 SEED1, A New Interpreter

We now present a new Polish grammar interpreter, SEED1, which is more efficient than SEED0. First, there are several new primitive functions needed:

Save : BOOLEANS x STRINGS(I) x STRINGS(O) ->
BOOLEANS x STRINGS(I) x STRINGS(O)
Save is the identity function, but it causes a triple to be stored in a special temporary location local to the procedure being executed.

Btemp : BOOLEANS
Itemp : STRINGS(I)
Otemp : STRINGS(O)
These three constants recover the temporary values stored by Save.

Boolean : BOOLEANS x STRINGS(I) x STRINGS(O) -> BOOLEANS
This function merely extracts the boolean component of a triple.

The following built-in function has been modified:

Output : STRINGS(O) x CHARACTERS -> STRINGS(O)
Output(S,C) appends C to S and returns the new S.

The SEED1 program (Fig. 8) is written in the SEEDGOL-1 language, which is an extension of SEEDGOL-0 and contains the new primitive functions described in this section, as well as CASE expressions and non-empty tuples of arbitrary length. In the new interpreter, SEED1, we no longer find the procedures Remaining or Emit. Skip and Find are just as in SEED0, except that a third parameter, which is not used by either function, is required by the syntax of the language.

```
IF BOOLEAN(SAVE(TEST(REST(GP), IP, OP))) THEN
   IF ISNULLI(ITEMP) THEN (TRUE, OTEMP)
   ELSE (FALSE, NULL)
ELSE (FALSE, NULL)
;
DEFINE TEST(RP, IP, OP) =
   CASE FIRST(RP) OF
   ":" : TEST(FIND(GP, REST(RP), OP), IP, OP)
   "&" : IF BOOLEAN(SAVE(TEST(REST(RP), IP, OP))) THEN
            IF BOOLEAN(SAVE(TEST(SKIP(REST(RP),IP,OP), ITEMP, OTEMP)))
               THEN (BTEMP, ITEMP, OTEMP)
            ELSE (FALSE, IP, OP)
         ELSE (FALSE, IP, OP)
   "/" : IF BOOLEAN(SAVE(TEST(REST(RP), IP, OP)))
            THEN (BTEMP, ITEMP, OTEMP)
         ELSE TEST(SKIP(REST(RP), IP, OP), IP, OP)
   ">" : (TRUE, IP, OUTPUT(OP, FIRST(REST(RP))))
   "#" : IF EQUAL(FIRST(REST(RP)), FIRST(IP))
            THEN (TRUE, REST(IP), OP)
         ELSE (FALSE, IP, OP)
   ENDCASE;

DEFINE SKIP(RP, IP, OP) =
   IF EQUAL(FIRST(RP), "&") THEN SKIP(SKIP(REST(RP), IP, OP), IP, OP)
   ELSE IF EQUAL(FIRST(RP), "/") THEN SKIP(SKIP(REST(RP),IP,OP),IP,OP)
   ELSE REST(REST(RP));

DEFINE FIND(RP, IP, OP) =
   IF EQUAL(FIRST(RP), FIRST(IP)) THEN REST(RP)
   ELSE FIND(SKIP(REST(RP), IP, OP), IP, OP);
END
```

SEED1, THE NEW GRAMMAR INTERPRETER, IN SEEDGOL-1

FIGURE 8

The functionality of SEED1 and its procedure Test are as follows:

SEED1 : STRINGS(G) x STRINGS(I) -> BOOLEANS x STRINGS(O)
SEED1(G,I) equals (TRUE, O) if and only if the input I is recognized by the object grammar G, in which case O is the output string. Otherwise, SEED1(G, I) equals (FALSE, NULL).

Test : STRINGS(G) x STRINGS(I) x STRINGS(O) -> BOOLEANS x STRINGS(I) x STRINGS(O)
Test(R, I, O) returns (B, I', O'), where B is TRUE if and only if a prefix of I is recognized by R, in which case I' is the tail of I remaining after recognition by R, and O' is the concatenation of O with the output characters encountered during recognition of I by R.

6. Implementing SEEDGOL-1

SEEDGOL-1 has restrictions similar to those of SEEDGOL-0:

1. As in SEEDGOL-0, all blanks are ignored, and no reserved word should be a prefix of an identifier or another reserved word.

2. There are no declarations other than procedure declarations. The only variables are the parameters RP, IP, and OP (strings), and the local temporary variables Btemp (boolean), Itemp, and Otemp (strings).

3. Procedures have exactly 3 parameters. Tuple elements and procedure arguments are evaluated left-to-right.

4. Procedures and mainline consist of a single expression.

5. The language has no I/O facility whatsoever. The user is expected to initialize the input and grammar strings, and retrieve the output string from its buffer.

6.1 Translating SEEDGOL-1 to Object Code

The translation of a SEEDGOL-1 program into object code for the SEEDGOL-1 machine, M1, can be effected by the

translation grammar SGL1G, shown in Figs. 9-A and 9-B. Fig. 10-B shows SEED1-O, the result of translating SEED1 (Fig. 8) via SGL1G. SEED1-O, when converted to binary form, implements SEED1 on M1. Fig. 10-A gives a mnemonic representation of SEED1-O.

6.2 M1, the SEEDGOL-1 Machine

The SEEDGOL-1 machine, M1, requires exactly the same storage areas and registers as the SEEDGOL-0 machine, M0 (Fig. 6), with the exception that the output buffer is no longer optional, but required, because the ability to reset the output pointer is needed. The main differences between the two machines are in the instruction set. Three instructions, OUTPUT, CALL, and RETURN, have been changed slightly in meaning, while PARM replaces IP and RP and handles the parameter OP and the local variables Btemp, Itemp, and Otemp as well. The four operators CASE, TEST, ENDTST, and ENDCAS implement both case and if-then-else expressions. POP is a new opcode, used twice by the "BOOLEAN" function (see Fig. 9-A) The new opcodes are summarized in Fig. 11. ALGOL and PDP-11 assembly language programs to implement M1 are shown in Appendices C and D.

The case expression operators are worthy of special comment. Suppose a SEEDGOL-1 program contains

```
CASE (exp0) OF
    (exp1) : (expA)
    (exp2) : (expB)
    (exp3) : (expC)
 ENDCASE  .
```

Then the object program would read:

```
CASE (exp0*) (exp1*) CASTST (expA*) ENDTST
             (exp2*) CASTST (expB*) ENDTST
             (exp3*) CASTST (expC*) ENDTST
 ENDCAS  .
```

The CASE instruction is a no-op used merely as a bracketing symbol. After exp0* and exp1* are executed, two values have been pushed on E. CASTST compares them, popping both if they are equal, and only the top (the value of exp1*) if they are not. Then, if they were equal, execution continues at CASTST + 1; otherwise execution continues at the next matching ENDTST + 1. (Here, matching is determined by counting TEST's and ENDTST's.) When an ENDTST is executed, the program counter is set to the next matching ENDCAS (with matching determined by counting CASE's and ENDCAS's).

```
P=E ";" [RETURN] D;
E="CASE" [CASE] E "OF" F
 /"IF" [CASE] E "THEN" [TRUE TEST] E ] "ELSE" [FALSE TEST] E
  [ENDTST ENDCAS]
 /"EQUAL(" E "," E ")" [EQUAL]
 /"ISNULLI(" E ")" [ISNULLI]
 /"REST(" E ")" [REST]
 /"OUTPUT(" E "," E ")" [OUTPUT]
 /"FIRST(" E ")" [FIRST]
 /"SAVE(" E ")" [SAVE]
 /"BOOLEAN(" E ")" [POP POP]
 /"BTEMP" [PARM 5]
 /"ITEMP" [PARM 4]
 /"OTEMP" [PARM 3]
 /"TRUE" [TRUE]/"FALSE" [FALSE]/"NULL"
 /"GP" [GP]/"RP" [PARM 0]/"IP" [PARM 1]/"OP" [PARM 2]
 /"(" E A
 /"""" S """"
 /[PROCN] I "(" E "," E "," E ")" [CALL]
 ;
A=")"
 /"," E A
 ;
F="ENDCASE" [ENDCAS]
 /E ":" [TEST] E [ENDTST] F
 ;
D="DEFINE" I "(RP,IP,OP)=" E ";" [RETURN] D
 /"END"
 ;
I=L I
 /L [ ]
 ;
S=":" [:]/"&" [&]/"/" [/]/">" [>]/"#" [#];
L="A" [A]/"B" [B]/"C" [C]/"D" [D]/"E" [E]/"F" [F]/"G" [G]
 /"H" [H]
 /"I" [I]/"J" [J]/"K" [K]/"L" [L]/"M" [M]/"N" [N]/"O" [O]
 /"P" [P]
 /"Q" [Q]/"R" [R]/"S" [S]/"T" [T]/"U" [U]/"V" [V]/"W" [W]
 /"X" [X]
 /"Y" [Y]/"Z" [Z]
 ;
END
```

SGL1G, THE SEEDGOL-1 TRANSLATION GRAMMAR
(MNEMONIC VERSION)

FIGURE 9-A

```
P=E ";" [I14] D;
E="CASE" [I8] E "OF" F
 /"IF" [I8] E "THEN" [D1I9] E [I10] "ELSE" [D0I9] E [I10I11]
 /"EQUAL(" E "," E ")" [I7]
 /"ISNULLI(" E ")" [I6]
 /"REST(" E ")" [I3]
 /"OUTPUT(" E "," E ")" [I4]
 /"FIRST(" E ")" [I2]
 /"SAVE(" E ")" [I15]
 /"BOOLEAN(" E ")" [I5I5]
 /"BTEMP" [I1D5]
 /"ITEMP" [I1D4]
 /"OTEMP" [I1D3]
 /"TRUE" [D1]/"FALSE" [D0]/"NULL"
 /"GP" [I0]/"RP" [I1D0]/"IP" [I1D1]/"OP" [I1D2]
 /"(" E A
 /""" S """
 /[I12] I "(" E "," E "," E ")" [I13]
 ;
A=")"
 /"," E A
 ;
F="ENDCASE" [I11]
 /E ":" [I9] E [I10] F
 ;
D="DEFINE" I "(RP,IP,OP)=" E ";" [I14] D
 /"END"
 ;
I=L I
 /L [" ]
 ;
S=":" [":]/"&" ["&]/"/" ["/]/">" [">]/"#" ["#];
L="A" ["A]/"B" ["B]/"C" ["C]/"D" ["D]/"E" ["E]/"F" ["F]/"G" ["G]
 /"H" ["H]/"I" ["I]/"J" ["J]/"K" ["K]/"L" ["L]/"M" ["M]/"N" ["N]
 /"O" ["O]/"P" ["P]/"Q" ["Q]/"R" ["R]/"S" ["S]/"T" ["T]/"U" ["U]
 /"V" ["V]/"W" ["W]/"X" ["X]/"Y" ["Y]/"Z" ["Z]
 ;
END
```

SGL1G, THE SEEDGOL-1 TRANSLATION GRAMMAR
(MACHINE VERSION)

FIGURE 9-B

001 TO 010:	CASE	PROCN	T	E	S	T		GP	REST	PARM
011 TO 020:	1	PARM	2	CALL	SAVE	POP	POP	1	TEST	CASE
021 TO 030:	PARM	4	ISNULL	1	TEST	1	PARM	3	ENDTST	0
031 TO 040:	TEST	0	ENDTST	ENDCAS	ENDTST	0	TEST	0	ENDTST	ENDCAS
041 TO 050:	RETURN	T	E	S	T		CASE	PARM	0	FIRST
051 TO 060:	:	TEST	PROCN	T	E	S	T		PROCN	F
061 TO 070:	I	N	D		GP	PARM	0	REST	PARM	2
071 TO 080:	CALL	PARM	1	PARM	2	CALL	ENDTST	&	TEST	CASE
081 TO 090:	PROCN	T	E	S	T		PARM	0	REST	PARM
091 TO 100:	1	PARM	2	CALL	SAVE	POP	POP	1	TEST	CASE
101 TO 110:	PROCN	T	E	S	T		PROCN	S	K	I
111 TO 120:	P		PARM	0	REST	PARM	1	PARM	2	CALL
121 TO 130:	PARM	4	PARM	3	CALL	SAVE	POP	POP	1	TEST
131 TO 140:	PARM	5	PARM	4	PARM	3	ENDTST	0	TEST	0
141 TO 150:	PARM	1	PARM	2	ENDTST	ENDCAS	ENDTST	0	TEST	0
151 TO 160:	PARM	1	PARM	2	ENDTST	ENDCAS	ENDTST	/	TEST	CASE
161 TO 170:	PROCN	T	E	S	T		PARM	0	REST	PARM
171 TO 180:	1	PARM	2	CALL	SAVE	POP	POP	1	TEST	PARM
181 TO 190:	5	PARM	4	PARM	3	ENDTST	0	TEST	PROCN	T
191 TO 200:	E	S	T		PROCN	S	K	I	P	
201 TO 210:	PARM	0	REST	PARM	1	PARM	2	CALL	PARM	1
211 TO 220:	PARM	2	CALL	ENDTST	ENDCAS	ENDTST	>	TEST	1	PARM
221 TO 230:	1	PARM	2	PARM	0	REST	FIRST	OUTPUT	ENDTST	#
231 TO 240:	TEST	CASE	PARM	0	REST	FIRST	PARM	1	FIRST	EQUAL
241 TO 250:	1	TEST	1	PARM	1	REST	PARM	2	ENDTST	0
251 TO 260:	TEST	0	PARM	1	PARM	2	ENDTST	ENDCAS	ENDTST	ENDCAS
261 TO 270:	RETURN	S	K	I	P		CASE	PARM	0	FIRST
271 TO 280:	&	EQUAL	1	TEST	PROCN	S	K	I	P	
281 TO 290:	PROCN	S	K	I	P		PARM	0	REST	PARM
291 TO 300:	1	PARM	2	CALL	PARM	1	PARM	2	CALL	ENDTST
301 TO 310:	0	TEST	CASE	PARM	0	FIRST	/	EQUAL	1	TEST
311 TO 320:	PROCN	S	K	I	P		PROCN	S	K	I
321 TO 330:	P		PARM	0	REST	PARM	1	PARM	2	CALL
331 TO 340:	PARM	1	PARM	2	CALL	ENDTST	0	TEST	PARM	0
341 TO 350:	REST	REST	ENDTST	ENDCAS	ENDTST	ENDCAS	RETURN	F	I	N
351 TO 360:	D		CASE	PARM	0	FIRST	PARM	1	FIRST	EQUAL
361 TO 370:	1	TEST	PARM	0	REST	ENDTST	0	TEST	PROCN	F
371 TO 380:	I	N	D		PROCN	S	K	I	P	
381 TO 390:	PARM	0	REST	PARM	1	PARM	2	CALL	PARM	1
391 TO 400:	PARM	2	CALL	ENDTST	ENDCAS	RETURN				

SEED1-O, THE M1 OBJECT CODE FOR SEED1
(MNEMONIC VERSION)

FIGURE 10-A

```
I8I12"T"E"S"T" I0I3I1D1I1D2I13I15I5I5D1I9I8I1D4I6D1I9D1I1D3I10D0I9D0I10I
11I10D0I9D0I10I11I14"T"E"S"T" I8I1D0I2":I9I12"T"E"S"T" I12"F"I"N"D" I0I1
D0I3I1D2I13I1D1I1D2I13I10"&I9IAI12"T"E"S"T" I1D0I3I1D1I1D2I13I15I5I5D1I9
I8I12"T"E"S"T" I12"S"K"I"P" I1D0I3I1D1I1D2I13I1D4I1D3I13I15I5I5D1I9I1D5I
1D4I1D3I10D0I9D0I1D1I1D2I10I11I10D0I9D0I1D1I1D2I10I11I10"/I9I8I12"T"E"S"
T" I1D0I3I1D1I1D2I13I15I5I5D1I9I1D5I1D4I1D3I10D0I9I12"T"E"S"T" I12"S"K"I
"P" I1D0I3I1D1I1D2I13I1D1I1D2I13I10I11I10">I9D1I1D1I1D2I1D0I3I2I4I10"#I9
I8I1D0I3I2I1D1I2I7D1I9D1I1D1I3I1D2I10D0I9D0I1D1I1D2I10I11I10I11I14"S"K"I
"P" I8I1D0I2"&I7D1I9I12"S"K"I"P" I12"S"K"I"P" I1D0I3I1D1I1D2I13I1D1I1D2I
13I10D0I9I8I1D0I2"/I7D1I9I12"S"K"I"P" I12"S"K"I"P" I1D0I3I1D1I1D2I13I1D1
I1D2I13I10D0I9I1D0I3I3I10I11I10I11I14"F"I"N"D" I8I1D0I2I1D1I2I7D1I9I1D0I
3I10D0I9I12"F"I"N"D" I12"S"K"I"P" I1D0I3I1D1I1D2I13I1D1I1D2I13I10I11I14
```

SEED1-O, THE M1 OBJECT CODE FOR SEED1
(MACHINE VERSION)

FIGURE 10-8

OPCODE	MNEMONIC	DESCRIPTION
(ANY CHARACTER)		(CHAR) PUSH THE OPCODE, A CHARACTER.
0	GP.	(STRING) PUSH POINTER TO FIRST CHARACTER IN G.
1	PARM	(ITEM) PUSH THE N-TH PARAMETER, WHERE N IS IN THE NEXT INSTRUCTION LOC (ADVANCE PC).
2	FIRST	(STRING -> CHAR) REPLACE TOP OF E WITH THE CHARACTER TO WHICH IT POINTS.
3	REST	(STRING -> STRING) INCREMENT PTR AT TOP OF E.
4	OUTPUT	(STRING × CHAR -> STRING) POP E, STORING THIS CHAR IN THE O LOCATION GIVEN BY TOP(E), WHICH IS THEN INCREMENTED.
5	POP	(ITEM ->) POP E.
6	ISNULLI	(STRING(I) -> BOOL) REPLACE TOP OF E WITH TRUE IF IT POINTS TO END OF I, ELSE FALSE.
7	EQUAL	(CHAR × CHAR -> BOOL) REPLACE THE TOP 2 CHARS ON E WITH TRUE IF THEY ARE EQUAL, ELSE FALSE.
8	CASE	( ) NO-OPERATION. MARKER ONLY.
9	TEST	(ITEM × ITEM -> ITEM OR NOTHING) IF THE TOP 2 ITEMS ON E ARE EQUAL, POP THEM BOTH, OTHERWISE, POP ONLY THE TOP, AND ADVANCE TO THE NEXT MATCHING ENDTST + 1.
10	ENDTST	( ) SKIP TO THE NEXT MATCHING ENDCAS + 1.
11	ENDCAS	( ) NO-OPERATION. MARKER ONLY.
12	PROCN	(ADDR) COMPARE THE CHARACTERS IN M AT PC WITH ALL STRINGS FOLLOWING RETURN INSTRUCTIONS ANYWHERE IN M. PUSH THE LOCATION IMMEDIATELY AFTER THE MATCHING STRING (THE CALLED ADDRESS). ADVANCE PC BEYOND THE PROCEDURE NAME.
13	CALL	(ADDR × ITEM × ITEM × ITEM ->) PUSH PC (RETURN ADDRESS) ONTO P, FOLLOWED BY THREE ZERO"S (TEMP STORAGE) AND THE THREE ITEMS POPPED FROM E. POP THE ADDR OFF E, AND BRANCH TO IT
14	RETURN	( ) IF P HAS 6 OR LESS ITEMS, SET THE HALT FLAG. OTHERWISE, POP THE 6 TOP ITEMS OFF P (3 TEMP VARIABLES, AND 3 PARAMETERS); SET PC TO THE ADDR POPPED OFF P NEXT.
15	SAVE	(ITEM × ITEM × ITEM -> ITEM × ITEM × ITEM) COPY THE TOP 3 ITEMS ON E INTO THE 3 CURRENT TEMP STORAGE LOCATIONS IN P.

ALL PUSHES AND POPS REFER TO THE E STACK, EXCEPT AS NOTED.
INSTRUCTIONS MUST BE MADE DISTINGUISHABLE FROM CONSTANTS.
THE SET "ITEM" IS THE UNION OF CHAR, STRING, AND BOOLEAN.

M1 MACHINE INSTRUCTIONS

FIGURE 11

If the value of exp0* does not equal the value of any expression before a CASTST, then control eventually drops down to the ENDCAS instruction, which is a no-op. Since none of the CASTST instructions popped the value of exp0*, the default value of the CASE expression is the value of the selector expression exp0*.

An IF-THEN-ELSE expression is translated into the code for a case expression in a straightforward manner:

```
IF (exp0) THEN (exp1) ELSE (exp2)
```

becomes

```
CASE (exp0*)  1 CASTST (exp1*) ENDTST
              0 CASTST (exp2*) ENDTST
 ENDCAS .
```

According to the conventions of the grammar SGL2G and the boolean-valued primitives of M1 (ISNULL and EQUAL), 1 is the value of TRUE, and 0 is equivalent to FALSE. It should be clear that the case operators above implement the IF-THEN-ELSE construct.

6.3 Initial Bootstrapping with M1

To get the SEED1 program running on M1, the user must:

1. Transcribe G1-O (Fig. 2), the object grammar grammar into machine readable form.

2. Transcribe SEED1-O (Fig. 10-B), the object code for SEED1 on M1.

3. Implement M1, the SEEDGOL-1 machine (Appendices C and D are examples).

4. Implement a means for getting prefix Polish "object" grammars, M1 instructions, and input strings into the storage areas for M1, and a means to retrieve the output string after translation.

7. Timing Results for Different Translators

The following table shows the timing results for the translation of G1 (Fig. 1) to G1-O (Fig. 2) using G1-O as the translation grammar.

Translator	Time
SEEDO, written in B5700 extended ALGOL	4 hours (**)
SEEDO, on M0 in B5700 extended ALGOL	154 hours (**)
SEEDO, on M0 in PDP-11 assembly language (**)	6 1/2 hours
SEED1, written in B5700 extended ALGOL	9 1/2 min. (25)
SEED1, on M1 in B5700 extended ALGOL	2 hours, 15 min. (68) (**) 1 hour, 20 min. (*)
SEED1, on M1 in PDP-11 assembly language	14 1/4 min. (27) 8 1/2 min. (*)

(*) -- estimate, when sped up using the PROCN2 modification below.
(**) -- estimate, based on partial runs and comparisons.

Numbers in parentheses indicate speedup factor over corresponding SEEDO implementation.

Some timing tests were run using a modified PROCN instruction, which would perform a linear search of the machine code storage area M only once to determine a called address. It would then store the called address in a table, along with the continuation address (the address at which execution continues, after the procedure name following the PROCN instruction). PROCN would store an index to the new table entries in the location following the PROCN instruction (which was formerly the first letter in the called procedure's name). Finally, it would change the PROCN instruction to a PROCN2 instruction, so that subsequent execution of the instruction would merely retrieve the called address and the continuation address from the table. This improvement yields a 40% speedup in execution, but requires care in implementation: any modifications to M must not create entries which may be construed as CASE, CASTST, ENDTST, or ENDCAS instructions. Because of this "trickiness" involved, the PROCN2 feature is not included in the M1 machines described in Fig. 11, or Appendices C and D. Implementation of this feature is left to the reader.

One unexpected anomaly which appears in Table 1 is the fact that the advantage of SEED1 over SEEDO is far greater on the B5700 in extended ALGOL then on the PDP-11 in assembly language. Apparently SEEDO requires extensive use of those sections of code which are particularly inefficient in ALGOL.

8. Conclusions

We have seen that a simple translator writing system with reasonable storage and CPU time requirements can be easily implemented without the aid of a compiler. The allowed translations are those which can be expressed via a context-free grammar, augmented with output symbols. Such translations can include renaming, and conversions between infix, prefix, and postfix notations. Address calculations, and other effects which cannot be expressed in a context-free grammar, cannot be performed by this simple syntax-driven method.

We observed by example that the simple translations described above can still be quite useful, since address calculations and other problems can be postponed until run time, and handled by an interpreter. We also observed that an interpreter for a very restricted (but still useful!) language can be quite simple, in spite of the need to calculate addresses. Important simplifying aspects of a language include such restrictions as a fixed number of parameters for any user-defined procedure, no user-declared variables, and a limited set of built-in functions.

The simple TWS given in this paper is a step in the evolution of a more sophisticated TWS. Hopefully, this simple TWS will facilitate the development of languages for describing scanners, parsers, and code generators, the constituents of typical TWS's.

It is a straightforward process to implement this simple system on other machines, including microprocessors.

Acknowledgements

Many of the ideas of this paper emerged during my conversations with Dan Ross, Doug Michels, and Tom Pennello. Bill McKeeman provided most of the impetus for this entire project. These and others (Frank Frazier, Bill Tyler, and Greg Johns) added many helpful comments during the preparation of this document. Doris Heinsohn prepared the tidy drawing of Fig. 6. Finally, my special thanks to Dan Ross for teaching me how to use the PDP-11.

REFERENCES

1. Abrahams, P. S. et al., The LISP2 programming language and system. Proc. AFIPS 1966 FJCC, 29, AFIPS Press, Montvale, N. J., 661-676.

2. Earley, J., An efficient context-free parsing algorithm. Comm. ACM 13, 2 (February 1970), 94-102.

3. Fay, M., GENESIS: A language for describing the bootstrapping process. Unpublished report, U. C. Santa Cruz (1976).

4. McKeeman, W. M., Private communication (1976).

5. Michels, D., A concise extensible metalanguage for translator implementation. Unpublished report, U. C. Santa Cruz (1976).

6. Naur, P. (ed.) et al., Report on the algorithmic language ALGOL 60. Comm. ACM 3, 5 (May 1960), 299-314.

7. PL/I Language Reference Manual. Form C28-8201-2. IBM Syst. Ref. Lib. (1969).

8. Schorre, D. V., A syntax oriented compiler writing language. 1964 ACM National Conference.

9. van Wijngaarden, A. et al., Revised report on the algorithmic language ALGOL 68. Acta Informatica 5 (1975), 1-236.

10. Wegbreit, Ben, The treatment of data types in EL1. Comm. ACM 17, 5 (May 1974), 251-264.

11. Wirth, N., Program development by stepwise refinement. Comm. ACM 14, 4 (April 1971), 221-227.

APPENDIX A

BURROUGHS B5700 EXTENDED ALGOL IMPLEMENTATION OF MO

```
BEGIN   % MO IN XALGOL

%DECLARATIONS, PRELIMINARIES

DEFINE TIL=STEP 1 UNTIL#;  % A TEXT SUBSTITUTION MACRO
ARRAY MCODE[0:1022];
ARRAY GI[0:1022];  % CHARACTER STORAGE
INTEGER ENDI;
INTEGER PC;  REAL INSTRUCTION;
INTEGER IPTR, GPTR;   % FIXED POINTERS TO G AND I IN CHAR STORAGE

% A STACK IS AN ARRAY WHOSE 0-TH ELEMENT POINTS TO THE TOP
DEFINE STACK = ARRAY#, TOP(S) = S[S[0]]#, STACKSIZE(S) = S[0]#,
       NEXT(S, I) = S[S[0] - (I)]#;
REAL PROCEDURE PUSH(X, S);  VALUE X;  STACK S[*];  REAL X;
   PUSH := S[S[0] := S[0] + 1] := X;      % WARNING:  NESTED ASSIGNMENT
REAL PROCEDURE POP(S);  STACK S[*];
   POP := S[(S[0]:=S[0]-1) + 1];      % WARNING:  NESTED ASSIGNMENT

STACK P [0:1022];    % PARAMETER STACK
STACK E [0:30];      % EVALUATION STACK
ARRAY O [0:1022];    % OUTPUT BUFFER

% X&SETINBIT SETS THE INSTRUCTION BIT OF X
% IS?INST(X) IS TRUE IF M IS AN INSTRUCTION, FALSE OTHERWISE
% X.OPFIELD IS THE OPCODE OF M, WITH THE INSTRUCTION BIT REMOVED
DEFINE SETINBIT = 1[7:0:1]#, INBIT = [7:1]#, INBITFIELD = [7:0:1]#;
DEFINE RET?INST = 13&SETINBIT#,
       IF?INST = 8&SETINBIT#, THEN?INST = 9&SETINBIT#,
       ELSE?INST = 10&SETINBIT#;

  % THE FOLLOWING MUST CORRESPOND WITH
  % CONSTANTS GENERATED BY THE GRAMMAR.
DEFINE FALSEVAL = 0#, TRUEVAL = 1#, BLANK = " "#;

DEFINE IS?INST(M) = ((M).INBIT = 1)#, OPFIELD = [6:7]#;
BOOLEAN HALT;
INTEGER CYCLES, TRACECNT;

PROCEDURE INITIALIZE;
   BEGIN
   O[0] := P[0] := E[0] := CYCLES := TRACECNT := 0;
   END INITIALIZE;

PROCEDURE GETINPUT;
   ;  % USER-PROVIDED ROUTINE TO LOAD GI (G & I) AND MCODE
      % THE MAIN ROUTINES EXPECT 1 CHARACTER OR OPCODE PER WORD

PROCEDURE SUMMARIZE;
   ;  % IF E[1] CONTAINS "TRUEVAL" THEN THE OUTPUT
      % STRING IS CONTAINED IN O[1] THROUGH O[STACKSIZE(O)],
      % ONE CHARACTER PER WORD.
```

```
% COMPLICATED INSTRUCTIONS

INTEGER REG1, REG2;

PROCEDURE SKIPPAST(INST);
      VALUE INST;  REAL INST;
   BEGIN
   REG1 := 0;
   WHILE REG1 GEQ 0 DO
      BEGIN
      IF MCODE[PC] = INST THEN REG1 := REG1 - 1
      ELSE IF MCODE[PC] = IF?INST THEN REG1 := REG1 + 1;
      PC := PC + 1;
      END;
   END SKIP PAST;

PROCEDURE CALL;
   BEGIN
   PUSH(PC, P);    % PUSH RETURN ADDRESS ON PARAMETER STACK.
   PUSH(POP(E), P);  % "IP", OR 2ND PARM
   PUSH(POP(E), P);  % "RP", OR 1ST PARM
   PC := POP(E);    % PROCNAME INSTR FIGURED OUT EXACT MACHINE ADDRESS
   END CALL;

PROCEDURE RETURN;
   IF STACKSIZE(P) LEQ 2 THEN HALT := TRUE
   ELSE
      BEGIN  POP(P);  POP(P);
      PC := POP(P);
      END;

PROCEDURE PROCNAME;
   BEGIN  REG2 := 1;    % 1 = START OF CODE
   DO
      BEGIN  REG1 := PC;  % START OF PROCEDURE NAME
      WHILE MCODE[REG2] NEQ RET?INST DO
         REG2 := REG2 + 1;
      REG2 := REG2 + 1;
      WHILE MCODE[REG1] NEQ BLANK AND MCODE[REG1] = MCODE[REG2] DO
         BEGIN  REG2 := REG2 + 1;
         REG1 := REG1 + 1;
         END
      END
   UNTIL MCODE[REG1] = MCODE[REG2];    % BOTH ARE " "
   PUSH(REG2 + 1, E);
   PC := REG1 + 1;
   END PROCNAME;
```

```
% MAIN EXECUTION LOOP OF INTERPRETER  * * *

DEFINE FETCH = INSTRUCTION := MCODE[(PC:=PC+1)-1]#;


PROCEDURE EXECUTE;
   IF NOT IS?INST(INSTRUCTION) THEN
      PUSH(MCODE[PC-1], E)     % CURRENT INSTRUCTION IS REALLY A
                               % CHARACTER; PUSH IT (CHAR)
   ELSE
      CASE INSTRUCTION.OPFIELD OF
         BEGIN
            % ZERO-ARY FUNCTIONS
         PUSH(GPTR, E);                  % 0, "GP" PTR TO 1ST OF G (PTR)
         PUSH(TOP(P), E);                % 1, "RP" REFERS TO P-STACK
                                     %  LOCATION POINTING TO G (PTR)
         PUSH(NEXT(P,1), E);             % 2, "IP", P-STACK PTR TO I (PTR)
            % UNARY FUNCTIONS
         PUSH(GI[POP(E)], E);            % 3, "FIRST" (CHAR)
         PUSH(POP(E) + 1, E);            % 4, "REST" (PTR -> PTR)
         PUSH(POP(E), O);                % 5, "OUTPUT" (CHAR -> ).
                                         %   LEAVES NO VALUE ON STACK, OUTPUT
                                         %   CHAR CANNOT BE RETRIEVED
         PUSH(IF POP(E) NEQ ENDI THEN FALSEVAL ELSE TRUEVAL, E);
                                         % 6, "ISNULLI" (PTR(I) -> BOOLEAN)
            % BINARY FUNCTION
         IF POP(E)=POP(E) THEN PUSH(TRUEVAL, E) ELSE PUSH(FALSEVAL, E);
                                         % 7, "EQUAL" (CHAR x CHAR -> BOOL)
            % CONTROL INSTRUCTIONS
         IF POP(E)=FALSEVAL THEN SKIPPAST(THEN?INST);
                                         % 8, "IF"
         SKIPPAST(ELSE?INST);            % 9, "THEN"
         ;                               % 10, "ELSE"
         PROCNAME;                       % 11, "PROCNAME"
         CALL;                           % 12, "CALL"
         RETURN                          % 13, "RETURN"
         END;


% M A I N L I N E  * * * * * * * * * * * * * * * *

INITIALIZE;
GETINPUT;
PUSH(IPTR, P);  % POINTER TO 1ST ELEMENT OF I  ("IP" PARAMETER)
PUSH(GPTR, P);  % POINTER TO 1ST ELEMENT OF G  ("RP" PARAMETER)
PC := 1;
HALT := FALSE;

DO
   BEGIN
   CYCLES := CYCLES + 1;
   FETCH;
   EXECUTE;
   END
UNTIL HALT;
SUMMARIZE;
END.
```

APPENDIX B

PDP-11 ASSEMBLY LANGUAGE IMPLEMENTATION OF MO

```
        .TITLE  MO
; MO, IN PDP-11 ASSEMBLY LANGUAGE

FILLIN = 0              ;VALUE TO BE FILLED IN AT RUN TIME
EPC = %0                ;EMULATOR PROGRAM COUNTER
REG1 = %1               ;TEMPORARY REG & INSTRUCTION BUFFER
REG2 = %2               ;TEMPORARY
E = %3                  ;EVALUATION STACK POINTER
P = %4                  ;PARAMETER STACK POINTER
PTRO = %5               ;PTR. TO 1ST POSITION AFTER END OF OUTPUT
SP = %6                 ;PDP-11 STACK POINTER
PC = %7                 ;PDP-11 PROGRAM COUNTER
R4 = %4
R5 = %5

FALSE = 0
TRUE = 1
IFX = 200+8.          ;SOME EMULATOR OPCODES
THENX = 200+9.        ;BIT 7 IS SET SO OPCODES AND CHARACTERS
ELSEX = 200+10.       ;CANNOT BE CONFUSED.
RETX = 200+13.
BLANK = " "     ;PROCEDURE NAME TERMINATOR

ESTACK: .BLKW 40.     ; EXECUTION STACK
ESTAK.:
PSTACK: .BLKW 1500. ; PARAMETER STACK, INCL RETURN ADDRS
PSTAK.:
STOP:   .WORD  FALSE  ;FOR TERMINATING EXECUTION LOOP

        .CSECT COM1 ; FORTRAN NAMED COMMON AREA
CYCLES: .WORD  0,0    ;DBL PREC COUNT OF EMULATOR CYCLES
I:      .BLKB  1000. ;INPUT STRING
EOI:
G:      .BLKB  1000. ;GRAMMAR (OBJECT VERSION) STRING
O:      .BLKB  2000. ;OUTPUT STRING
M:      .BLKB  1000. ;EMULATOR CODE
LENI:   .WORD  FILLIN;  NO. OF CHARS ACTUALLY IN I
LENO:   .WORD  FILLIN;  NO. OF CHARS IN O
PARSED: .WORD  FILLIN;  BOOLEAN VALUE. TRUE IF SUCCESSFUL PARSE

        .CSECT  IFACE  ;INTERFACE TO FORTRAN CODE (SAVE R4, R5)
IFACE:  MOV R4, SAVER4 ;CALLING SEQUENCE:  CALL IFACE
        MOV R5, SAVER5
        JSR PC, MO  ;EXECUTE THE EMULATOR
        MOV SAVER5, R5
        MOV SAVER4, R4
        RTS R5          ;RETURN TO DOS FORTRAN
SAVER4: .WORD  FILLIN
SAVER5: .WORD  FILLIN
```

```
        .CSECT        ;THE EMULATOR FOR SEEDGOL-1
;  INITIALIZATION OF EMULATOR
M0:     MOV #M, EPC    ;SET PROG COUNTER TO START OF CODE
        MOV #ESTAK., E  ;E-STACK POINTER
        MOV #PSTAK., P  ;P-STACK POINTER
        MOV #O,PTRO     ;NEXT OUTPUT SPACE IS THE FIRST
        MOV #I, -(P)    ;PUSH(PTR(I), P)  ("IP" PARAMETER)
        MOV #G, -(P)    ;PUSH(PTR(G), P)  ("RP" PARAMETER)
;  FETCH-EXECUTE LOOP
LOOP:   INC CYCLES
        CMP CYCLES,#10000.
        BLT L1
        INC CYCLES+2
        CLR CYCLES
L1:     MOVB (EPC)+, REG1    ;FETCH INSTRUCTION, ADVANCE EPC
        BIC #177400, REG1    ;INSTRUCTION IS ONLY ONE BYTE
        BIT #200, REG1       ;OPCODE IF BIT 7 IS ON
        BNE CASE
        MOV REG1,-(E)        ;OTHERWISE, A LITERAL CALL
        BR LOOP              ;(PUSH 1 WORD WITH O"S IN HIGH BYTE)
CASE:   BIC #200, REG1       ;MASK OFF INSTRUCTION BIT
        ASL REG1             ;FOR WORD ADDRESSING
        JSR PC, @OPS(REG1)   ;BRANCH TO AN INSTRUCTION ROUTINE
        TST STOP             ;CHECK HALTING FLAG
        BEQ LOOP
        MOV (E), PARSED      ;TOP OF E INDICATES SUCCESS IF TRUE
        MOV PTRO, LENO
        SUB #O, LENO
        RTS PC

;  ADDRESSES OF THE EMULATION PROCEDURES FOR THE OPCODES
OPS:    .WORD  GP, RP, IP, FIRST
        .WORD  REST, OUTPUT, ISNULL, EQUAL, IF, THEN
        .WORD  ELSE, PROCN, CALL, RETURN

;  THE EMULATION PROCEDURES FOR THE OPCODES
GP:     MOV #G, -(E)         ;PUSH(PTR(G), E)
        RTS PC
RP:     MOV (P), -(E)        ;PUSH(TOP(P), E)  ("RP" PARAMETER)
        RTS PC
IP:     MOV 2(P), -(E)       ;PUSH(NEXT(P,1), E)   ("IP" PARAMETER)
        RTS PC
FIRST:  MOVB @(E), (E)         ;PUSH(CONTENTS(POP(E)), E)
        BIC #177400, (E)     ;TOP BYTE IS ALL O"S, NOW
        RTS PC
REST:   INC (E)              ;TOP(E) := TOP(E) + 1
        RTS PC
OUTPUT: MOV (E)+, REG1       ;PUSH(POP(E), O)
        MOVB REG1, (PTRO)+   ;WORD TO BYTE CONVERSION
        RTS PC
ISNULL: SUB LENI, (E)        ;PUSH(POP(E) = ENDOF(I), E)
        SUB #I, (E)
        BEQ ISNUL2
        CLR (E)              ;NEQ => FALSE
        RTS PC
ISNUL2: MOV #TRUE, (E)
```

```
        RTS PC
EQUAL:  CMP (E)+, (E)       ;PUSH(POP(E)=POP(E), E)
        BEQ EQUAL2          ;2-BYTE COMPARISON ; SINGLE BYTE ITEMS
        CLR (E)             ;MUST BE EXPANDED TO WORDS CONSISTENTLY
        RTS PC
EQUAL2: MOV #TRUE,(E)
        RTS PC
IF:     TST (E)+            ;SKIP TO MATCHING "THEN" IF
        BNE IF2             ;POP(E) IS FALSE, ELSE DO NOTHING
        MOV #THENX, REG2
        JSR PC, SKIPTO
IF2:    RTS PC
THEN:   MOV #ELSEX, REG2    ;SKIP TO MATCHING "ELSE"
        JSR PC, SKIPTO
ELSE:   RTS PC
PROCN:  MOV #M, REG2        ;REG2 WILL POINT TO CHARS AFTER RETX
PROCN1: CMPB (REG2)+, #RETX
        BNE PROCN1
        MOV EPC, REG1       ;REG1 WILL POINT TO CHARS AFTER PROCN
PROCN2: CMPB (REG1)+, (REG2)+
        BNE PROCN1          ;FIND NEXT "RETURN" IF MISMATCH
        CMPB #BLANK,-1(REG2)
        BNE PROCN2          ;CONTINUE UNLESS BOTH ARE BLANK
        MOV REG2, -(E)      ;PUSH(CALLING ADDR, E)
        MOV REG1, EPC       ;EPC := LOC AFTER NAME
        RTS PC
CALL:   MOV EPC, -(P)       ;PUSH(EPC, P)   (RETURN ADDR)
        MOV (E)+, -(P)      ;PUSH(POP(E), P)   (R-PARAMETER)
        MOV (E)+, -(P)      ;PUSH(POP(E), P)   (I-PARAMETER)
        MOV (E)+, EPC       ;EPC := POP(E)  (CALLING ADDR)
        RTS PC
RETURN: CMP (P)+, (P)+      ;POP 2 WORDS OFF PSTACK
        CMP P, #PSTAK.      ;HALT IF P-STACK EMPTY
        BNE RET2
        MOV #TRUE, STOP
        RTS PC
RET2:   MOV (P)+, EPC       ;EPC := RETURN ADDRESS (=POP(P))
        RTS PC

SKIPTO: CLR REG1            ;THIS ROUTINE USED BY "IF" AND "THEN"
SKIP2:  CMPB (EPC), REG2    ;ADVANCE EPC, INCREMENTING REG1 ON "IF",
        BEQ SKIP1           ;DECREMENTING REG1 ON SOUGHT INSTRUCTION
        CMPB (EPC)+, #IFX   ;(CONTAINED IN REG2).
        BNE SKIP2           ;QUIT WHEN REG1 IS LESS THAN 0
        INC REG1
        BR SKIP2
SKIP1:  INC EPC
        DEC REG1
        BGE SKIP2
        RTS PC
        .END
```

APPENDIX C

BURROUGHS B5700 EXTENDED ALGOL IMPLEMENTATION OF M1

```
BEGIN   % M1 IN XALGOL

%DECLARATIONS, PRELIMINARIES

DEFINE TIL=STEP 1 UNTIL#;  % A TEXT SUBSTITUTION MACRO
ARRAY MCODE[0:1022];       % MACHINE CODE
ARRAY GI[0:1022];          % CHARACTER STORAGE
INTEGER ENDI;
INTEGER PC;  REAL INSTRUCTION;
INTEGER IPTR, GPTR;   % FIXED POINTERS TO G AND I IN CHAR STORAGE

% A STACK IS AN ARRAY WHOSE 0-TH ELEMENT POINTS TO THE TOP
DEFINE STACK = ARRAY#, TOP(S) = S[S[0]]#, STACKSIZE(S) = S[0]#,
       NEXT(S, I) = S[S[0] - (I)]#;
REAL PROCEDURE PUSH(X, S);  VALUE X;  STACK S[*];  REAL X;
   PUSH := S[S[0] := S[0] + 1] := X;      % WARNING:  NESTED ASSIGNMENT
REAL PROCEDURE POP(S);  STACK S[*];
   POP := S[(S[0]:=S[0]-1) + 1];      % WARNING:  NESTED ASSIGNMENT

STACK P [0:1022];    % PARAMETER STACK
STACK E [0:30];      % EVALUATION STACK
ARRAY O [0:1022];    % OUTPUT BUFFER

% X&SETINBIT SETS THE INSTRUCTION BIT OF X
% IS?INST(X) IS TRUE IF M IS AN INSTRUCTION, FALSE OTHERWISE
% X.OPFIELD IS THE OPCODE OF M, WITH THE INSTRUCTION BIT REMOVED
DEFINE SETINBIT = 1[7:0:1]#, INBIT = [7:1]#, INBITFIELD = [7:0:1]#;
DEFINE RET?INST = 14&SETINBIT#, CASE?INST = 8&SETINBIT#,
       TEST?INST = 9&SETINBIT#, ENDTST?INST = 10&SETINBIT#,
       ENDCAS?INST = 11&SETINBIT#;

   % THE FOLLOWING MUST CORRESPOND WITH
   % CONSTANTS GENERATED BY THE GRAMMAR.
DEFINE FALSEVAL = 0#, TRUEVAL = 1#, BLANK = " "#;

DEFINE IS?INST(M) = ((M).INBIT = 1)#, OPFIELD = [6:7]#;
BOOLEAN HALT;
INTEGER CYCLES, TRACECNT;

PROCEDURE INITIALIZE;
   BEGIN
   O[0] := P[0] := E[0] := CYCLES := TRACECNT := 0;
   END INITIALIZE;

PROCEDURE GETINPUT;
   ;  % USER-PROVIDED ROUTINE TO LOAD GI (G & I) AND MCODE
      % THE MAIN ROUTINES EXPECT 1 CHARACTER OR OPCODE PER WORD

PROCEDURE SUMMARIZE;
   ;  % IF E[1] CONTAINS "TRUEVAL" THEN THE OUTPUT
      % STRING IS CONTAINED IN O[1] THROUGH O[E[2]],
      % ONE CHARACTER PER WORD.
```

```
% COMPLICATED INSTRUCTIONS

INTEGER REG1, REG2;

PROCEDURE SKIPPAST(INST1, INST2);
        VALUE INST1, INST2;  REAL INST1, INST2;
    BEGIN
    REG1 := 0;
    WHILE REG1 GEQ 0 DO
        BEGIN
        IF MCODE[PC] = INST1 THEN REG1 := REG1 + 1
        ELSE IF MCODE[PC] = INST2 THEN REG1 := REG1 - 1;
        PC := PC + 1;
        END;
    END;

PROCEDURE CALL;
    BEGIN
    PUSH(PC, P);    % PUSH RETURN ADDRESS ON PARAMETER STACK.
    THRU 3 DO PUSH(0, P);
    THRU 3 DO PUSH(POP(E), P);  % OP, IP, AND RP PARAMETERS
    PC := POP(E);    % PROCNAME INSTR FIGURED OUT EXACT MACHINE ADDRESS
    END CALL;

PROCEDURE RETURN;
    IF STACKSIZE(P) LEQ 6 THEN HALT := TRUE
    ELSE
        BEGIN  THRU 6 DO POP(P);
        PC := POP(P);
        END;

PROCEDURE PROCNAME;
    BEGIN  REG2 := 1;    % 1 = START OF CODE
    DO
        BEGIN  REG1 := PC;  % START OF PROCEDURE NAME
        WHILE MCODE[REG2] NEQ RET?INST DO
            REG2 := REG2 + 1;
        REG2 := REG2 + 1;
        WHILE MCODE[REG1] NEQ BLANK AND MCODE[REG1] = MCODE[REG2] DO
            BEGIN  REG2 := REG2 + 1;
            REG1 := REG1 + 1;
            END
        END
    UNTIL MCODE[REG1] = MCODE[REG2];    % BOTH ARE BLANK
    PUSH(REG2 + 1, E);
    PC := REG1 + 1;
    END PROCNAME;
```

```
% MAIN EXECUTION LOOP OF INTERPRETER  * * *

DEFINE FETCH = INSTRUCTION := MCODE[(PC:=PC+1)-1]#;

PROCEDURE EXECUTE;
   BEGIN  INTEGER I;
   IF NOT IS?INST(INSTRUCTION) THEN
      PUSH(INSTRUCTION, E)     % CURRENT INSTRUCTION IS REALLY A
                               % CHARACTER; PUSH IT (CHAR)
   ELSE
      CASE INSTRUCTION.OPFIELD OF
         BEGIN
            % ZERO-ARY FUNCTIONS
         PUSH(GPTR, E);                   % 0, "GP".  PTR TO 1ST CHAR OF G
                                          %   AND TO 1 PAST END OF I (PTR)
         PUSH(NEXT(P, MCODE[(PC:=PC+1)-1]), E);
                                          % 1, "PARM" GET N-TH PARM, WHERE
                                          %   N IS IN NEXT INST LOC
            % UNARY FUNCTIONS
         PUSH(GI[POP(E)], E);             % 2, "FIRST" (CHAR)
         PUSH(POP(E) + 1, E);             % 3, "REST" (PTR -> PTR)
         O[NEXT(E,1):=NEXT(E,1)+1] := POP(E);
                                          % 4, "OUTPUT" (PTR x CHAR -> PTR).
         POP(E);                          % 5, "POP" (ITEM -> )
         PUSH(IF POP(E)=ENDI THEN TRUEVAL ELSE FALSEVAL, E);
                                          % 6, "ISNULLI"
            % BINARY FUNCTION
         IF POP(E)=POP(E) THEN PUSH(TRUEVAL, E) ELSE PUSH(FALSEVAL, E);
                                          % 7, "EQUAL" (CHAR x CHAR -> BOOL)
            % CONTROL INSTRUCTIONS
         ;                                % 8, "CASE"
         IF POP(E) NEQ TOP(E) THEN SKIPPAST(TEST?INST, ENDTST?INST)
         ELSE POP(E);
                                          % 9, "TEST"
         SKIPPAST(CASE?INST, ENDCAS?INST);
                                          % 10, "ENDTST"
         ;                                % 11, "ENDCAS"
         PROCNAME;                        % 12, "PROCNAME"
         CALL;                            % 13, "CALL"
         RETURN;                          % 14, "RETURN"
         FOR I := 0 TIL 2 DO NEXT(P, 3+I) := NEXT(E, I);
                                          % 15, "SAVE"
         END;
   END EXECUTE;
```

```
% M A I N L I N E * * * * * * * * * * * * * * *

INITIALIZE;
GETINPUT;
THRU 3 DO PUSH(O, P);     % TEMPORARY STORAGE LOCATIONS
PUSH(O, P);       % POINTER TO 1ST ELEMENT OF O ("OP" PARAMETER)
PUSH(IPTR, P);  % POINTER TO 1ST ELEMENT OF I  ("IP" PARAMETER)
PUSH(GPTR, P);  % POINTER TO 1ST ELEMENT OF G  ("RP" PARAMETER)
PC := 1;
HALT := FALSE;

DO
   BEGIN
   CYCLES := CYCLES + 1;
   FETCH;
   EXECUTE;
   END
UNTIL HALT;
SUMMARIZE;
END.
```

APPENDIX D

PDP-11 ASSEMBLY LANGUAGE IMPLEMENTATION OF M1

```
        .TITLE  M1
; M1, IN PDP-11 ASSEMBLY LANGUAGE

FILLIN = 0              ;VALUE TO BE FILLED IN AT RUN TIME
EPC = %0                ;EMULATOR PROGRAM COUNTER
REG1 = %1               ;TEMPORARY REG & INSTRUCTION BUFFER
REG2 = %2               ;TEMPORARY
E = %3                  ;EVALUATION STACK POINTER
P = %4                  ;PARAMETER STACK POINTER
PTRO = %5               ;PTR. TO 1ST POSITION AFTER END OF OUTPUT
SP = %6                 ;PDP-11 STACK POINTER
PC = %7                 ;PDP-11 PROGRAM COUNTER
R4 = %4
R5 = %5

FALSE = 0
TRUE = 1
TESTX = 200+9.                  ;SOME EMULATOR OPCODES
ENDTSX = 200+10.                ;BIT 7 IS SET SO OPCODES AND CHARACTERS
CASEX = 200+8.                  ;CANNOT BE CONFUSED
ENDCAX = 200+11.
RETX = 200+14.
BLANK = " "     ;PROCEDURE NAME TERMINATOR

ESTACK: .BLKW 40.    ; EXECUTION STACK
ESTAK.:
PSTACK: .BLKW 3000. ; PARAMETER STACK, INCL RETURN ADDRS
PSTAK.:
STOP:   .WORD  FALSE  ;FOR TERMINATING EXECUTION LOOP

        .CSECT COM1 ; FORTRAN NAMED COMMON AREA
CYCLES: .WORD  0,0   ;DBL PREC COUNT OF EMULATOR CYCLES
I:      .BLKB  1000. ;INPUT STRING
EOI:
G:      .BLKB  1000. ;GRAMMAR (OBJECT VERSION) STRING
O:      .BLKB  2000. ;OUTPUT STRING
M:      .BLKB  1000. ;EMULATOR CODE
LENI:   .WORD  FILLIN;  NO. OF CHARS ACTUALLY IN I
LENO:   .WORD  FILLIN;  NO. OF CHARS IN O
PARSED: .WORD  FILLIN;  BOOLEAN VALUE. TRUE IF SUCCESSFUL PARSE

        .CSECT  IFACE  ;INTERFACE TO FORTRAN CODE (SAVE R4, R5)
IFACE:  MOV R4, SAVER4 ;CALLING SEQUENCE:  CALL IFACE
        MOV R5, SAVER5
        JSR PC, MO  ;EXECUTE THE EMULATOR
        MOV SAVER5, R5
        MOV SAVER4, R4
        RTS R5          ;RETURN TO DOS FORTRAN
SAVER4: .WORD  FILLIN
SAVER5: .WORD  FILLIN
```

```
        .CSECT        ;THE EMULATOR FOR SEEDGOL-1
;  INITIALIZATION OF EMULATOR
M0:     MOV #M, EPC    ;SET PROG COUNTER TO START OF CODE
        MOV #ESTAK., E  ;E-STACK POINTER
        MOV #PSTAK.-6, P  ;P-STACK POINTER WITH 3 TEMP LOCATIONS
        MOV #O,-(P)      ;PUSH(PTR(O), P)  ("OP" PARAMETER)
        MOV #I, -(P)     ;PUSH(PTR(I), P)  ("IP" PARAMETER)
        MOV #G, -(P)     ;PUSH(PTR(G), P)  ("RP" PARAMETER)
;  FETCH-EXECUTE LOOP
LOOP:   INC CYCLES
        CMP CYCLES,#10000.
        BLT L1
        INC CYCLES+2
        CLR CYCLES
L1:     MOVB (EPC)+, REG1    ;FETCH INSTRUCTION, ADVANCE EPC
        BIC #177400, REG1     ;INSTRUCTION IS ONLY ONE BYTE
        BIT #200, REG1       ;OPCODE IF BIT 7 IS ON
        BNE CASES
        MOV REG1,-(E)        ;OTHERWISE, A LITERAL CALL
        BR LOOP              ;(PUSH 1 WORD WITH 0"S IN HIGH BYTE)
CASES:  BIC #200, REG1       ;MASK OFF INSTRUCTION BIT
        ASL REG1              ;FOR WORD ADDRESSING
        JSR PC, @OPS(REG1)   ;BRANCH TO AN INSTRUCTION ROUTINE
        TST STOP              ;CHECK HALTING FLAG
        BEQ LOOP
        MOV 2(E), PARSED     ;NEXT OF E INDICATES SUCCESS IF TRUE
        MOV (E), LENO    ;TOP OF E POINTS TO NEXT AVAIL SPACE IN O
        SUB #O, LENO
        RTS PC

;  ADDRESSES OF THE EMULATION PROCEDURES FOR THE OPCODES
OPS:    .WORD  GP, PARM, FIRST, REST, OUTPUT
        .WORD  POP, ISNULL, EQUAL, CASE, TEST
        .WORD  ENDTST, ENDCAS, PROCN, CALL, RETURN
        .WORD  SAVE

;  THE EMULATION PROCEDURES FOR THE OPCODES
GP:     MOV #G, -(E)          ;PUSH(PTR(G), E)
        RTS PC
PARM:   MOVB (EPC)+,REG1 ;GET PARM AT N-TH FROM TOP OF P,
        ASL REG1          ;(DOUBLE OFFSET FOR WORD ADDRESSING)
        ADD P,REG1        ;N IS IN NEXT CODE LOCATION
        MOV (REG1),-(E) ;PUSH RESULT ON E
        RTS PC
FIRST:  MOVB @(E), (E)          ;PUSH(CONTENTS(POP(E)), E)
        BIC #177400, (E)   ;TOP BYTE IS ALL 0"S, NOW
        RTS PC
REST:   INC (E)              ;TOP(E) := TOP(E) + 1
        RTS PC
OUTPUT: MOV (E)+,REG1    ;O[NEXT(E,1)] := POP(E)
        MOVB REG1,@(E)
        INC (E)           ;TOP(E) := TOP(E)+1
        RTS PC
POP:    CMP (E)+,(E)     ;POP(E)
        RTS PC
ISNULL: SUB LENI, (E)      ;PUSH(POP(E) = ENDOF(I), E)
```

```
        SUB #1, (E)
        BEQ ISNUL2
        CLR (E)             ;NEQ => FALSE
        RTS PC
ISNUL2: MOV #TRUE, (E)
        RTS PC
EQUAL:  CMP (E)+, (E)       ;PUSH(POP(E)=POP(E), E)
        BEQ EQUAL2          ;2-BYTE COMPARISON ; SINGLE BYTE ITEMS
        CLR (E)             ;MUST BE EXPANDED TO WORDS CONSISTENTLY
        RTS PC
EQUAL2: MOV #TRUE,(E)
CASE:   RTS PC          ;NO-OP
TEST:   CMP (E)+,(E)    ;IF POP(E) NEQ TOP(E) THEN SKIP PAST
        BEQ TEST2       ;NEXT MATCHING ENDTST INSTRUCTION
        MOV #TESTX,INST1        ;ELSE POP(E)
        MOV #ENDTSX,INST2
        JSR PC,SKIPTO
        RTS PC
TEST2:  CMP (E)+,(E)
        RTS PC
ENDTST: MOV #CASEX,INST1        ;SKIP PAST MACHING ENDCAS INSTR
        MOV #ENDCAX,INST2
        JSR PC, SKIPTO
ENDCAS: RTS PC                  ;A NO-OP
PROCN:  MOV #M, REG2      ;REG2 WILL POINT TO CHARS AFTER RETX
PROCN1: CMPB (REG2)+, #RETX
        BNE PROCN1
        MOV EPC, REG1     ;REG1 WILL POINT TO CHARS AFTER PROCN
PROCN2: CMPB (REG1)+, (REG2)+
        BNE PROCN1        ;FIND NEXT "RETURN" IF MISMATCH
        CMPB #BLANK,-1(REG2)
        BNE PROCN2        ;CONTINUE UNLESS BOTH ARE BLANK
        MOV REG2,-(E)   ;PUSH(CALLED ADDR, E)
        MOV REG1,EPC    ;EPC := LOC AFTER NAME
        RTS PC
CALL:   MOV EPC,-(P)    ;PUSH(EPC, P)   (RETURN ADDR)
        SUB #6,P        ;PUSH 3 TEMP LOCATIONS
        MOV (E)+,-(P)   ;PUSH(POP(E), P)  (OP-PARAMETER)
        MOV (E)+,-(P)   ;PUSH(POP(E), P)   (R-PARAMETER)
        MOV (E)+,-(P)   ;PUSH(POP(E), P)   (I-PARAMETER)
        MOV (E)+,EPC    ;EPC := POP(E)  (CALLING ADDR)
        RTS PC
RETURN: ADD #12.,P      ;POP 6 WORDS OFF PSTACK
        CMP P, #PSTAK.  ;HALT IF P-STACK EMPTY
        BNE RET2
        MOV #TRUE, STOP
        RTS PC
RET2:   MOV (P)+, EPC     ;EPC := RETURN ADDRESS (=POP(P))
        RTS PC

SAVE:   MOV (E),6(P)    ;STORE THE TOP 3 THINGS ON E IN THE
        MOV 2(E),8.(P)  ;TEMP LOCATIONS IN THE P-STACK
        MOV 4(E),10.(P) ;DOES NOT AFFECT E
        RTS PC
```

```
INST1:   .WORD    ;PARAMETERS FOR SKIPTO SUBROUTINE
INST2:   .WORD
SKIPTO:  CLR REG1              ;THIS ROUTINE USED BY "TEST" AND "ENDTST"
SKIP2:   CMPB (EPC), INST2     ;ADVANCE EPC, INCREMENTING REG1 ON INST1,
         BEQ SKIP1             ;DECREMENTING REG1 ON SOUGHT INSTRUCTION
         CMPB (EPC)+, INST1    ;(CONTAINED IN INST2).
         BNE SKIP2             ;QUIT WHEN REG1 IS LESS THAN 0
         INC REG1
         BR SKIP2
SKIP1:   INC EPC
         DEC REG1
         BGE SKIP2
         RTS PC
         .END
```

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER 14 TR- Tech. Rep. No. 77-3-ØØ2	2. GOVT ACCESSION NO.
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 Bootstrapping a Small Translator Writing System.	5. ~~TYPE OF REPORT & PERIOD COVERED~~ 9 Technical [redacted] rept.
	6. PERFORMING ~~ORG.~~ REPORT NUMBER
7. AUTHOR(s) 10 Michael Fay	8. ~~CONTRACT OR GRANT NUMBER(s)~~ 15 N ØØØ14-76-C-Ø682
9. PERFORMING ORGANIZATION NAME AND ADDRESS William M. McKeeman Calif Univ ~~Information Sciences~~, UCSC, Rm. 239 AS Santa Cruz, Ca. 95064	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, Virginia 22217	12. REPORT DATE 11 28 Mar[redacted] 1977
	13. NUMBER OF PAGES 51 12 55p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research University of California 553 Evans Hall Berkeley, California 94720	15. SECURITY CLASS. ~~(of this report)~~
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

388073

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

metalanguage, translator, syntax-directed translation, translator writing system, self-describing grammar, interpreter, bootstrapping, backtrack parsing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A rudimentary translator writing system is developed for easy implementation in about 2 pages of assembly language code. Although the system is based on backtrack parsing and lacks a scanner, it still performs useful translations in a few minutes of CPU time, with storage requirements of about 10K bytes, for a typical translation.

(continued on reverse side)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

per JB 388Ø73

The system is based on an ALGOL-like program by Michels which translates source language strings into target language strings, according to a translation grammar which is specified using prefix Polish operators. Fortunately, the user does not need to specify translation grammars in Polish notation, because Michels gave a metagrammar which translates grammars in BNF-like notation (including the metagrammar itself) into Polish strings.

This report shows how Michels' program can be implemented without the aid of an ALGOL compiler. We present a translation grammar for converting Michels' program (slightly rewritten) into code for a simple, special-purpose interpreter. Once this simple interpreter is implemented, and Michels' program (in interpreter code) and the first input grammar are prepared, a small translator writing system is complete. In this primitive system, a translator "program" consists of the BNF-like description of a translation grammar.

Michels' program was written with the goal of conceptual simplicity. However, in actual performance it was found to be too slow to be practical. Accordingly, we present a new program which is shorter, more efficient, and which requires only a slightly more complex interpreter.

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0682

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Code 102IP
Arlington, VA 22217
6 copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area Office
715 Broadway - 5th Floor
New York, NY 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, DC 20375
6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (CodeRD-1)
Washington, D. C. 20380
1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-911G)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy